Lightning Aura Components Developer Guide

Version 63.0, Spring '25

Last updated: March 18, 2025

CONTENTS

CHAPTER 1 Introducing Aura Components

In this chapter ...

Lightning components is the umbrella term for Aura components and Lightning web components. As of Spring '19 (API version 45.0), you can build Lightning components using two programming models: the Lightning Web Components (LWC) model, and the original Aura Components model.

The Lightning Component framework is a UI framework for developing Lightning components for mobile and desktop devices. Lightning web components and Aura components can coexist and interoperate on a page.

EDITIONS

Available in: Lightning Experience

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

Lightning Web Components uses core Web Components standards and provides only what's necessary to perform well in browsers supported by Salesforce. Because it's built on code that runs natively in browsers, Lightning Web Components is lightweight and delivers exceptional performance. Most of the code you write is standard JavaScript and HTML.

For new components, create Lightning web components instead of Aura components. Lightning web components perform better and are easier to develop than Aura components. However, when you develop Lightning web components, you also may need to use Aura, because LWC doesn't yet support everything that Aura does. We're actively working in each release to eliminate these gaps so that LWC works for all use cases.

Configure Lightning web components and Aura components to work in Lightning App Builder and Experience Builder. Admins and end users don't know which programming model was used to develop the components. To them, they're simply Lightning components.

This developer guide covers how to develop custom Aura components. The Lightning Web Components Developer Guide covers how to develop custom Lightning web components.

Tip: The name of the programming model is Aura Components (uppercase). When we refer to the components themselves, we use Aura components (lowercase).

What is Salesforce Lightning?

Lightning includes the Lightning Component Framework and some exciting tools for developers. Lightning makes it easier to build responsive applications for any device.

Lightning includes these technologies:

- Lightning components accelerate development and app performance. Develop custom components that other developers and admins can use as reusable building blocks to customize Lightning Experience and the Salesforce mobile app.
- Lightning App Builder empowers admins to build Lightning pages visually, without code, using off-the-shelf and custom-built Lightning components. Make your Lightning components available in the Lightning App Builder so administrators can build custom user interfaces without code.
- Experience Builder empowers admins to build communities visually, without code, using Lightning templates and components. Make your Lightning components available in Experience Builder so administrators can build community pages without code.

Using these technologies, you can seamlessly customize and easily deploy new apps to mobile devices running Salesforce. In fact, the Salesforce mobile app and Salesforce Lightning Experience are built with Lightning components.

This guide teaches you to create your own custom Aura components and apps. You also learn how to package applications and components and distribute them in the AppExchange.

To learn how to develop Lightning web components, see Lightning Web Components Developer Guide.

Use Lightning Web Components instead of Aura Components

Lightning web components perform better and are easier to develop than Aura components. However, when you develop Lightning web components, you also may need to use Aura, because LWC doesn't yet support everything that Aura does.

How do you decide which components to develop as Lightning web components and which to develop as Aura components?

The answer is to always choose Lightning Web Components unless you need a feature that isn't supported.

For information on gaps between Lightning Web Components and Aura Components, see the Lightning Web Components Developer Guide.

To migrate Aura components to Lightning web components, see the Lightning Web Components Developer Guide.

Aura Components Release Notes

Use the Salesforce Release Notes to learn about the most recent updates and changes to Aura Components.

For updates and changes that impact Aura Components, see Lightning Components in the Salesforce Release Notes.

For new and changed Aura components, see Lightning Components: New and Changed Items in the Salesforce Release Notes.

Aura Components

Aura components are the self-contained and reusable units of an app. They represent a reusable section of the UI, and can range in granularity from a single line of text to an entire app.

Important: Creating Aura components isn't supported in Starter and Pro Suite Editions.

The framework includes a set of prebuilt components. For example, components that come with the Lightning Design System styling are available in the `lightning` namespace. These components are also known as the base Lightning components. You can assemble and configure components to form new components in an app. Components are rendered to produce HTML DOM elements within the browser.

A component can contain other components, as well as HTML, CSS, JavaScript, or any other Web-enabled code. This enables you to build apps with sophisticated UIs.

The details of a component's implementation are encapsulated. This allows the consumer of a component to focus on building their app, while the component author can innovate and make changes without breaking consumers. You configure components by setting the named attributes that they expose in their definition. Components interact with their environment by listening to or publishing events.

SEE ALSO:

Creating Components

Working with Base Lightning Components

Events

Event-driven programming is used in many languages and frameworks, such as JavaScript and Java Swing. The idea is that you write handlers that respond to interface events as they occur.

A component registers that it may fire an event in its markup. Events are fired from JavaScript controller actions that are typically triggered by a user interacting with the user interface.

There are two types of events in the framework:

- **Component events** are handled by the component itself or a component that instantiates or contains the component.
- **Application events** are handled by all components that are listening to the event. These events are essentially a traditional publish-subscribe model.

You write the handlers in JavaScript controller actions.

SEE ALSO:

Communicating with Events

Handling Events with Client-Side Controllers

Browser Support for Aura Components

Aura Components support the same browsers as Lightning Experience.

For more information, see Supported Browsers for Lightning Experience.

SEE ALSO:

Salesforce Help: Recommendations and Requirements for all Browsers

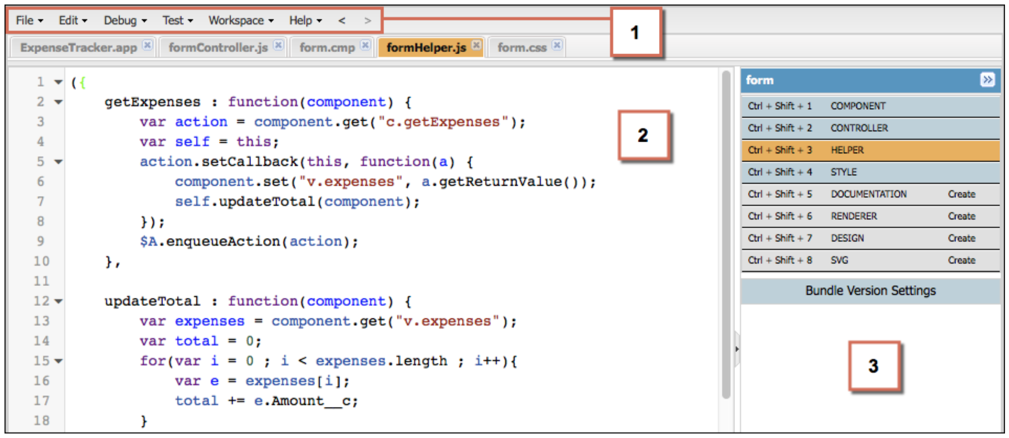
Security for Lightning Components: Lightning Locker Disabled for Unsupported Browsers

Security for Lightning Components: Content Security Policy Overview

Using the Developer Console

The Developer Console provides tools for developing your Aura components and applications.

You can use the Developer Console in the same supported browsers as Lightning Experience and Salesforce Classic.


File
Edit
Debug
Test
Workspace
Help
1
ExpenseTracker.app
formController.js
form.cmp
formHelper.js
form.css
2
form
Ctrl + Shift + 1 COMPONENT
Ctrl + Shift + 2 CONTROLLER
Ctrl + Shift + 3 HELPER
Ctrl + Shift + 4 STYLE
Ctrl + Shift + 5 DOCUMENTATION Create
Ctrl + Shift + 6 RENDERER Create
Ctrl + Shift + 7 DESIGN Create
Ctrl + Shift + 8 SVG Create
Bundle Version Settings
3

```
({
    getExpenses : function(component) {
        var action = component.get("c.getExpenses");
        var self = this;
        action.setCallback(this, function(a) {
            component.set("v.expenses", a.getReturnValue());
            self.updateTotal(component);
        });
        $A.enqueueAction(action);
    },

    updateTotal : function(component) {
        var expenses = component.get("v.expenses");
        var total = 0;
        for(var i = 0 ; i < expenses.length ; i++){
            var e = expenses[i];
            total += e.Amount__c;
        }
```

The Developer Console enables you to perform these functions.

- Use the menu bar (1) to create or open these Lightning resources.
  - Application
  - Component
  - Interface
  - Event
  - Tokens
- Use the workspace (2) to work on your Lightning resources.
- Use the sidebar (3) to create or open client-side resources that are part of a specific component bundle.
  - Controller
  - Helper
  - Style
  - Documentation
  - Renderer
  - Design
  - SVG

While the Developer Console provides an easy way to work with Aura components, it doesn't include many developer tools and features. To enable source-drive development with editor features like code completion and linting, consider these alternatives:

- Code Builder—A web-based IDE that has all the power and flexibility of VS Code, Salesforce Extensions for VS Code, and Salesforce CLI in your web browser. You can install Code Builder as a managed package in a supported Salesforce org edition.

- Salesforce DX Tools—Use the Salesforce CLI and VS Code with the Salesforce Extension Pack to deploy code to an org.

SEE ALSO:

Salesforce Help: Open the Developer Console

Create Aura Components in the Developer Console

Component Bundles

Online Content

This guide is available online. To view the latest version, go to:

`https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/`

Go beyond this guide with exciting Trailhead content. To explore more of what you can do with Lightning Components, go to:

Trailhead Module: Lightning Components Basics

Link: https://trailhead.salesforce.com/module/lex_dev_lc_basics

Learn with a series of hands-on challenges on how to use Lightning Components to build modern web apps.

Quick Start: Lightning Components

Link: https://trailhead.salesforce.com/project/quickstart-lightning-components

Create your first component that renders a list of Contacts from your org.

Project: Build an Account Geolocation App

Link: https://trailhead.salesforce.com/project/account-geolocation-app

Build an app that maps your Accounts using Lightning Components.

Project: Build a Restaurant-Locator Lightning Component

Link: https://trailhead.salesforce.com/project/workshop-lightning-restaurant-locator

Build a Lightning component with Yelp’s Search API that displays a list of businesses near a certain location.

Project: Build a Lightning App with the Lightning Design System

Link: https://trailhead.salesforce.com/project/slds-lightning-components-workshop

Design a Lightning component that displays an Account list.

CHAPTER 2 Quick Start

In this chapter ...

The quick start provides Trailhead resources for you to learn core Aura components concepts, and a short tutorial that builds an Aura component to manage selected contacts in the Salesforce mobile app and Lightning Experience. You'll create all components from the Developer Console. The tutorial uses several events to create or edit contact records, and view related cases.

Before You Begin

To work with Lightning apps and components, create a Developer Edition org.

Note: For this quick start tutorial, you don't need to create a Developer Edition organization or register a namespace prefix. But you want to do so if you're planning to offer managed packages. You can create Aura components using the UI in **Enterprise**, **Performance**, **Unlimited**, **Developer** Editions, or a sandbox.

You need an org to do this quick start tutorial, and we recommend that you don't use your production org. You only need to create a Developer Edition org if you don't already have one.

1. In your browser, go to `https://developer.salesforce.com/signup?d=70130000000td6N`.
2. Fill in the fields about you and your company.
3. In the `Email` field, make sure to use a public address you can easily check from a Web browser.
4. Type a unique `Username`. Note that this field is also in the *form* of an email address, but it does not have to be the same as your email address, and in fact, it's usually better if they aren't the same. Your username is your login and your identity on `developer.salesforce.com`, so you're often better served by choosing a username such as `firstname@lastname.com`.
5. Read and then select the checkbox for the `Main Services Agreement` and then click **Submit Registration**.
6. In a moment you'll receive an email with a login link. Click the link and change your password.

Trailhead: Explore Lightning Aura Components Resources

Learn the fundamentals of Lightning Aura components with Trailhead resources.

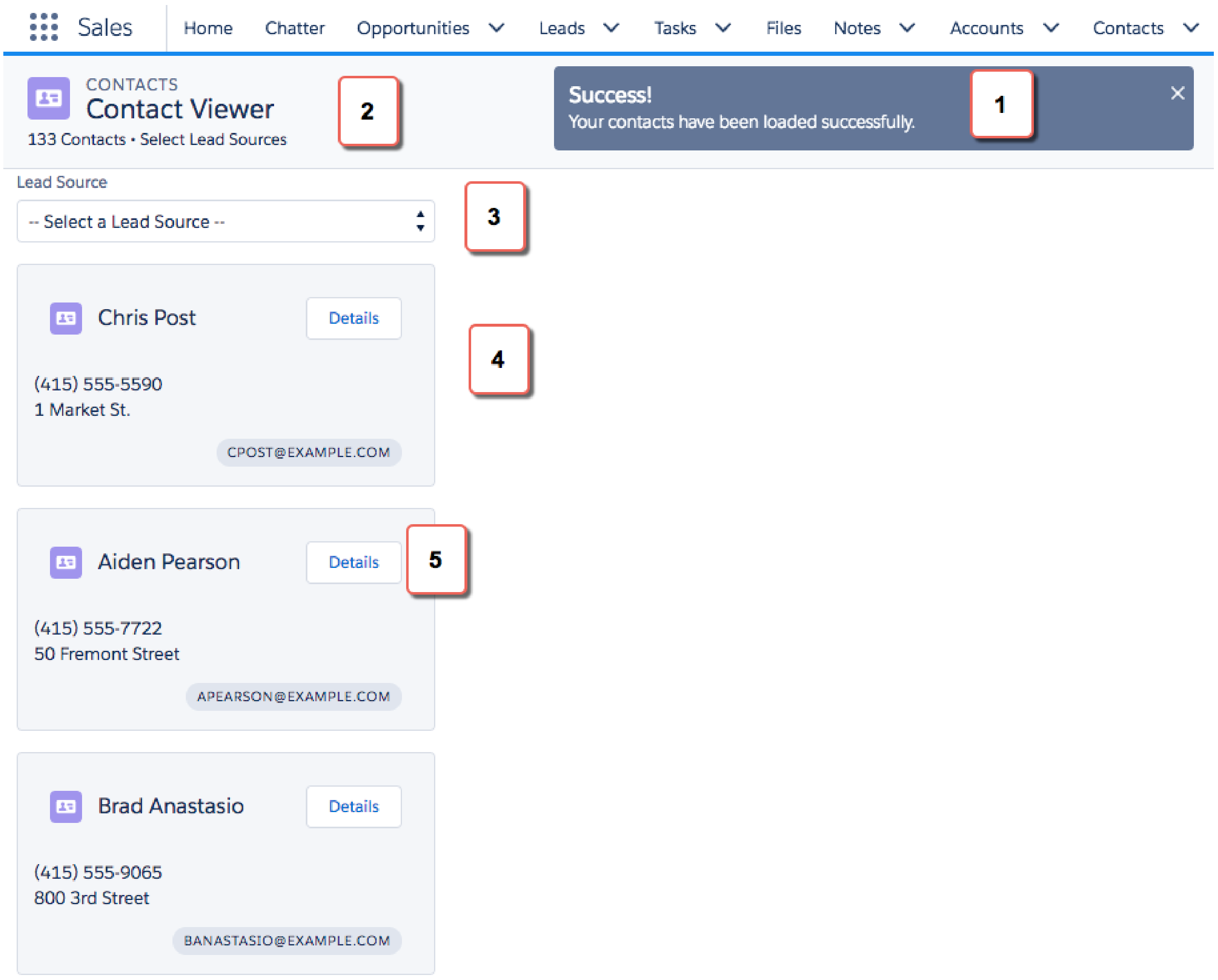
Whether you're a new or seasoned Salesforce developer, we recommend that you start with the following Trailhead resource: Quick Start: Aura Components.

Create a Component for Lightning Experience and the Salesforce Mobile App

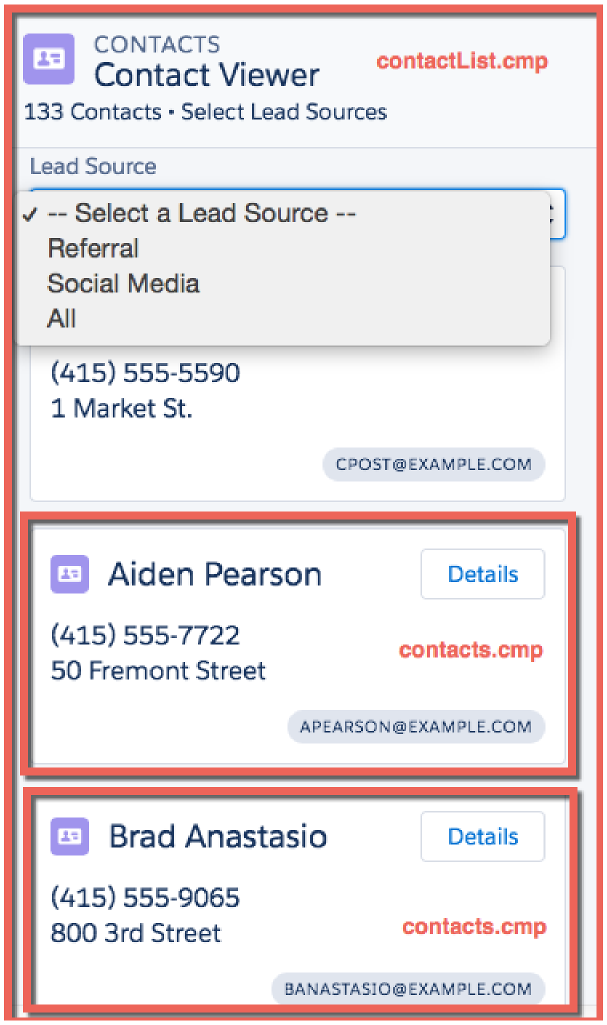
Explore how to create a custom UI that loads contact data and interacts with Lightning Experience and the Salesforce mobile app.

This tutorial walks you through creating a component that:

- Displays a toast message (1) using the `force:showToast` event when all contacts are loaded successfully.
- Updates the number of contacts (2) based on the selected lead source.
- Filters the contacts using the `lightning:select` component (3) when a lead source (referral or social media) is selected.
- Displays the contact data using the `lightning:card` component (4).
- Navigates to the record when the **Details** button (5) is clicked.



Here's how the component looks in the Salesforce mobile app. You're creating two components, `contactList` and `contacts`, where `contactList` is a container component that iterates over and displays `contacts` components. All contacts are displayed in `contactList`, but you can select different lead sources to view a subset of contacts associated with the lead source.



In the next few topics, you create the following resources.

Resource	Description
Contacts Bundle	
`contacts.cmp`	The component that displays individual contacts
`contactsController.js`	The client-side controller action that navigates to a contact record using the `force:navigateToSObject` event
contactList Bundle	
`contactList.cmp`	The component that loads the list of contacts
`contactListController.js`	The client-side controller actions that call the helper resource to load contact data and handles the lead source selection
`contactListHelper.js`	The helper function that retrieves contact data, displays a toast message on successful loading of contact data, displays contact data based on lead source, and update the total number of contacts
Apex Controller	

Resource	Description
`ContactController.apxc`	The Apex controller that queries all contact records and those records based on different lead sources

Load the Contacts

Create an Apex controller and load your contacts. An Apex controller is the bridge that connects your components and your Salesforce data.

Your organization must have existing contact records for this tutorial.

1. In the Developer Console, click **File** > **New** > **Apex Class**, and then enter *`ContactController`* in the **New Class** window. A new Apex class, `ContactController.apxc`, is created. Enter this code and then save.

```
public with sharing class ContactController {
@AuraEnabled
    public static List<Contact> getContacts() {
        List<Contact> contacts =
               [SELECT Id, Name, MailingStreet, Phone, Email, LeadSource FROM Contact];


        //Add isAccessible() check
        return contacts;
    }
}
```

   `ContactController` contains methods that return your contact data using SOQL statements. This Apex controller is wired up to your component in a later step. `getContacts()` returns all contacts with the selected fields.

2. Click **File** > **New** > **Lightning Component**, and then enter *`contacts`* for the `Name` field in the New Lightning Bundle popup window. This creates a component, `contacts.cmp`. Enter this code and then save.

```
<aura:component>
    <aura:attribute name="contact" type="Contact" />

        <lightning:card variant="Narrow" title="{!v.contact.Name}"
                        iconName="standard:contact">
            <aura:set attribute="actions">
              <lightning:button name="details" label="Details" onclick="{!c.goToRecord}"
 />
            </aura:set>
            <aura:set attribute="footer">
                <lightning:badge label="{!v.contact.Email}"/>
            </aura:set>
            <p class="slds-p-horizontal_small">
                {!v.contact.Phone}
            </p>
            <p class="slds-p-horizontal_small">
                {!v.contact.MailingStreet}
            </p>
        </lightning:card>

</aura:component>
```

This component creates the template for your contact data using the `lightning:card` component, which simply creates a visual container around a group of information. This template gets rendered for every contact that you have, so you have multiple instances of a component in your view with different data. The `onclick` event handler on the `lightning:button` component calls the `goToRecord` client-side controller action when the button is clicked. Notice the expression `{!v.contact.Name}`? `v` represents the view, which is the set of component attributes, and `contact` is the attribute of type `Contact`. Using this dot notation, you can access the fields in the contact object, like `Name` and `Email`, after you wire up the Apex controller to the component in the next step.

3. Click **File** > **New** > **Lightning Component**, and then enter *`contactList`* for the `Name` field in the New Lightning Bundle popup window, which creates the `contactList.cmp` component. Enter this code and then save. If you're using a namespace in your organization, replace `ContactController` with `myNamespace.ContactController`. You wire up the Apex controller to the component by using the `controller="ContactController"` syntax.

```
<aura:component implements="force:appHostable" controller="ContactController">
    <!-- Handle component initialization in a client-side controller -->
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <!-- Dynamically load the list of contacts -->
    <aura:attribute name="contacts" type="Contact[]"/>
    <aura:attribute name="contactList" type="Contact[]"/>
    <aura:attribute name="totalContacts" type="Integer"/>

    <!-- Page header with a counter that displays total number of contacts -->
    <div class="slds-page-header slds-page-header_object-home">
        <lightning:layout>
            <lightning:layoutItem>
                <lightning:icon iconName="standard:contact" />
            </lightning:layoutItem>
            <lightning:layoutItem class="slds-m-left_small">
                <p class="slds-text-title_caps slds-line-height_reset">Contacts</p>
                <h1 class="slds-page-header__title slds-p-right_x-small">Contact
Viewer</h1>
            </lightning:layoutItem>
        </lightning:layout>

        <lightning:layout>
            <lightning:layoutItem>
                <p class="slds-text-body_small">{!v.totalContacts} Contacts • View
Contacts Based on Lead Sources</p>
            </lightning:layoutItem>
        </lightning:layout>
    </div>

    <!-- Body with dropdown menu and list of contacts -->
    <lightning:layout>
        <lightning:layoutItem padding="horizontal-medium" >
            <!-- Create a dropdown menu with options -->
            <lightning:select aura:id="select" label="Lead Source" name="source"
                              onchange="{!c.handleSelect}" class="slds-m-bottom_medium">

                <option value="">-- Select a Lead Source --</option>
                <option value="Referral" text="Referral"/>
                <option value="Social Media" text="Social Media"/>
                <option value="All" text="All"/>
            </lightning:select>

            <!-- Iterate over the list of contacts and display them -->
            <aura:iteration var="contact" items="{!v.contacts}">
                <!-- If you're using a namespace, replace with myNamespace:contacts-->
                <c:contacts contact="{!contact}"/>
            </aura:iteration>
        </lightning:layoutItem>
    </lightning:layout>
</aura:component>
```

Let's dive into the code. We added the `init` handler to load the contact data during initialization. The handler calls the client-side controller code in the next step. We also added two attributes, `contacts` and `totalContacts`, which stores the list of contacts and a counter to display the total number of contacts respectively. Additionally, the `contactList` component is an attribute used to store the filtered list of contacts when an option is selected on the lead source dropdown menu. The `lightning:layout` components simply create grids to align your content in the view with Lightning Design System CSS classes.

The page header contains the `{!v.totalContacts}` expression to dynamically display the number of contacts based on the lead source you select. For example, if you select **Referral** and there are 30 contacts whose `Lead Source` fields are set to `Referral`, then the expression evaluates to 30.

Next, we create a dropdown menu with the `lightning:select` component. When you select an option in the dropdown menu, the `onchange` event handler calls your client-side controller to update the view with a subset of the contacts. You create the client-side logic in the next few steps.

In case you're wondering, the `force:appHostable` interface enables your component to be surfaced in Lightning Experience and the Salesforce mobile app as tabs, which we are getting into later.

4. In the **contactList** sidebar, click **CONTROLLER** to create a resource named `contactListController.js`. Replace the placeholder code with the following code and then save.

```
({
    doInit : function(component, event, helper) {
        // Retrieve contacts during component initialization
        helper.loadContacts(component);
    },

    handleSelect : function(component, event, helper) {
        var contacts = component.get("v.contacts");
        var contactList = component.get("v.contactList");

        //Get the selected option: "Referral", "Social Media", or "All"
        var selected = event.getSource().get("v.value");

        var filter = [];
        var k = 0;
        for (var i=0; i<contactList.length; i++){
            var c = contactList[i];
            if (selected != "All"){
                if(c.LeadSource == selected) {
                    filter[k] = c;
                    k++;
                }
            }
            else {
                    filter = contactList;
               }
          }
          //Set the filtered list of contacts based on the selected option
          component.set("v.contacts", filter);
          helper.updateTotal(component);
     }
})
```

The client-side controller calls helper functions to do most of the heavy-lifting, which is a recommended pattern to promote code reuse. Helper functions also enable specialization of tasks, such as processing data and firing server-side actions, which is what we are covering next. Recall that the `onchange` event handler on the `lightning:select` component calls the `handleSelect` client-side controller action, which is triggered when you select an option in the dropdown menu. `handleSelect` checks the option value that's passed in using `event.getSource().get("v.value")`. It creates a filtered list of contacts by checking that the lead source field on each contact matches the selected lead source. Finally, update the view and the total number of contacts based on the selected lead source.

5. In the **contactList** sidebar, click **HELPER** to create a resource named `contactListHelper.js`. Replace the placeholder code with the following code and then save.

```
({
   loadContacts : function(cmp) {
         // Load all contact data
         var action = cmp.get("c.getContacts");
         action.setCallback(this, function(response) {
             var state = response.getState();
             if (state === "SUCCESS") {
                 cmp.set("v.contacts", response.getReturnValue());
                 cmp.set("v.contactList", response.getReturnValue());
                 this.updateTotal(cmp);
             }

             // Display toast message to indicate load status
             var toastEvent = $A.get("e.force:showToast");
             if (state === 'SUCCESS'){
                 toastEvent.setParams({
                     "title": "Success!",
                     "message": " Your contacts have been loaded successfully."
                 });
             }
             else {
                 toastEvent.setParams({
                         "title": "Error!",
                         "message": " Something has gone wrong."
                 });
             }
             toastEvent.fire();
         });
          $A.enqueueAction(action);
     },

     updateTotal: function(cmp) {
       var contacts = cmp.get("v.contacts");
       cmp.set("v.totalContacts", contacts.length);
```

```
    }
})
```

During initialization, the `contactList` component loads the contact data by:

- Calling the Apex controller method `getContacts`, which returns the contact data via a SOQL statement
- Setting the return value via `cmp.set("v.contacts", response.getReturnValue())` in the action callback, which updates the view with the contact data
- Updating the total number of contacts in the view, which is evaluated in `updateTotal`

You must be wondering how your component works in Lightning Experience and the Salesforce app. Let's find out next!

6. Make the `contactList` component available via a custom tab in Lightning Experience and the Salesforce app.
   - Add Aura Components as Custom Tabs in a Lightning Experience App

   For this tutorial, we recommend that you add the component as a custom tab in Lightning Experience.

When your component is loaded in Lightning Experience or the Salesforce app, a toast message indicates that your contacts are loaded successfully. Select a lead source from the dropdown menu and watch your contact list and the number of contacts update in the view.

Next, wire up an event that navigates to a contact record when you click a button in the contact list.

Fire the Events

Fire the events in your client-side controller or helper functions. The `force` events are handled by Lightning Experience and the Salesforce mobile app, but let's view and test the components in Lightning Experience to simplify things.

This demo builds on the contacts component you created in Load the Contacts on page 10.

1. In the **contacts** sidebar, click **CONTROLLER** to create a resource named `contactsController.js`. Replace the placeholder code with the following code and then save.

```
({
    goToRecord : function(component, event, helper) {
        // Fire the event to navigate to the contact record
        var sObjectEvent = $A.get("e.force:navigateToSObject");
        sObjectEvent.setParams({
            "recordId": component.get("v.contact.Id")
        })
        sObjectEvent.fire();
    }
})
```

   The `onclick` event handler in the following button component triggers the `goToRecord` client-side controller when the button is clicked.

```
<lightning:button name="details" label="Details" onclick="{!c.goToRecord}" />
```

   You set the parameters to pass into the events using the `event.setParams()` syntax. In this case, you're passing in the Id of the contact record to navigate to. There are other events besides `force:navigateToSObject` that simplify navigation within Lightning Experience and the Salesforce app. For more information, see Events Handled in the Salesforce Mobile App and Lightning Experience.

2. To test the event, refresh your custom tab in Lightning Experience, and click the **Details** button.

   The `force:navigateToSObject` is fired, which updates the view to display the contact record page.

We stepped through creating a component that loads contact data using a combination of client-side controllers and Apex controller methods to create a custom UI with your Salesforce data. The possibilities of what you can do with Aura components are endless. While we showed you how to surface a component via a tab in Lightning Experience and the Salesforce app, you can take this tutorial further by surfacing the component on record pages via the Lightning App Builder and even Experience Builder. To explore the possibilities, blaze the trail with the resources available at Trailhead: Explore Lightning Aura Components Resources.

CHAPTER 3 Creating Components

In this chapter ...

Components are the functional units of the Lightning Component framework.

Important: Creating Aura components isn't supported in Starter and Pro Suite Editions.

A component encapsulates a modular and potentially reusable section of UI, and can range in granularity from a single line of text to an entire application.

- Using Labels
- Localization
- Working with Base Lightning Components
- Supporting Accessibility
- Writing Documentation for the Component Library

Component Names

A component name must follow the naming rules for Lightning components.

A component name must follow these naming rules:

- Must begin with a letter
- Must contain only alphanumeric or underscore characters
- Must be unique in the namespace
- Can't include whitespace
- Can't end with an underscore
- Can't contain two consecutive underscores

SEE ALSO:

Create Aura Components in the Developer Console

Component Markup

Create Aura Components in the Developer Console

The Developer Console is a convenient, built-in tool you can use to create new and edit existing Aura components and other bundles.

1. Open the Developer Console.

   Select **Developer Console** from the *Your Name* or the quick access menu (⚙).

2. Open the New Lightning Bundle panel for an Aura component.

   Select **File** > **New** > **Lightning Component**.

3. Name the component.

   For example, enter `helloWorld` in the Name field.

4. Optional: Describe the component.

   Use the Description field to add details about the component.

5. Optional: Add component configurations to the new component.

   You can select as many options in the Component Configuration section as you wish, or select no configuration at all.

6. Click **Submit** to create the component.

   Or, to cancel creating the component, click the panel's close box in the top right corner.

IN THIS SECTION:

Lightning Bundle Configurations Available in the Developer Console
Configurations make it easier to create a component or application for a specific purpose, like a Lightning page or Experience Builder site page, or a quick action or navigation item in Lightning Experience or Salesforce mobile app. The New Lightning Bundle panel in the Developer Console offers a choice of component configurations when you create an Aura component or application bundle.

SEE ALSO:

Using the Developer Console
Lightning Bundle Configurations Available in the Developer Console
Create Aura Components Using Salesforce CLI

Lightning Bundle Configurations Available in the Developer Console

Configurations make it easier to create a component or application for a specific purpose, like a Lightning page or Experience Builder site page, or a quick action or navigation item in Lightning Experience or Salesforce mobile app. The New Lightning Bundle panel in the Developer Console offers a choice of component configurations when you create an Aura component or application bundle.

Configurations add the interfaces required to support using the component in the desired context. For example, when you choose the **Lightning Tab** configuration, your new component includes `implements="force:appHostable"` in the `<aura:component>` tag.

Using configurations is optional. You can use them in any combination, including all or none.

The following configurations are available in the New Lightning Bundle panel.

Configuration	Markup	Description
Aura component bundle		
Lightning Tab	`implements="force:appHostable"`	Creates a component for use as a navigation element in Lightning Experience or Salesforce mobile apps.
Lightning Page	`implements="flexipage:availableForAllPageTypes"` and `access="global"`	Creates a component for use in Lightning pages or the Lightning App Builder.
Lightning Record Page	`implements="flexipage:availableForRecordHome, force:hasRecordId"` and `access="global"`	Creates a component for use on a record home page in Lightning Experience.
Experience Builder Site Page (previously Lightning Communities Page)	`implements="forceCommunity:availableForAllPageTypes"` and `access="global"`	Creates a component that's available for drag and drop in the Experience Builder.
Lightning Quick Action	`implements="force:lightningQuickAction"`	Creates a component that can be used with a Lightning quick action.
Lightning application bundle		

Configuration	Markup	Description
Lightning Out Dependency App	`extends="ltng:outApp"`	Creates an empty Lightning Out dependency app.

Note: For details of the markup added by each configuration, see the respective documentation for those features.

SEE ALSO:

- Create Aura Components in the Developer Console
- Configure Components for Custom Tabs
- Configure Components for Custom Actions
- Configure Components for Lightning Pages and the Lightning App Builder
- Configure Components for Lightning Experience Record Pages
- Configure Components for Experience Builder

Create Aura Components Using Salesforce CLI

To develop Aura components, use Salesforce CLI to synchronize source code between your Salesforce orgs and version control system. Alternatively, you can use the Developer Console.

Your development environment includes:

- Salesforce CLI
- Visual Studio Code or another code editor
- Salesforce Extension Pack, if using Visual Studio Code
- A Developer Edition org

To install Salesforce CLI and verify the installation, follow the instructions at Salesforce CLI Setup Guide.

Note: If you have an old version of the CLI installed, run this command to update it.

```
sf update
```

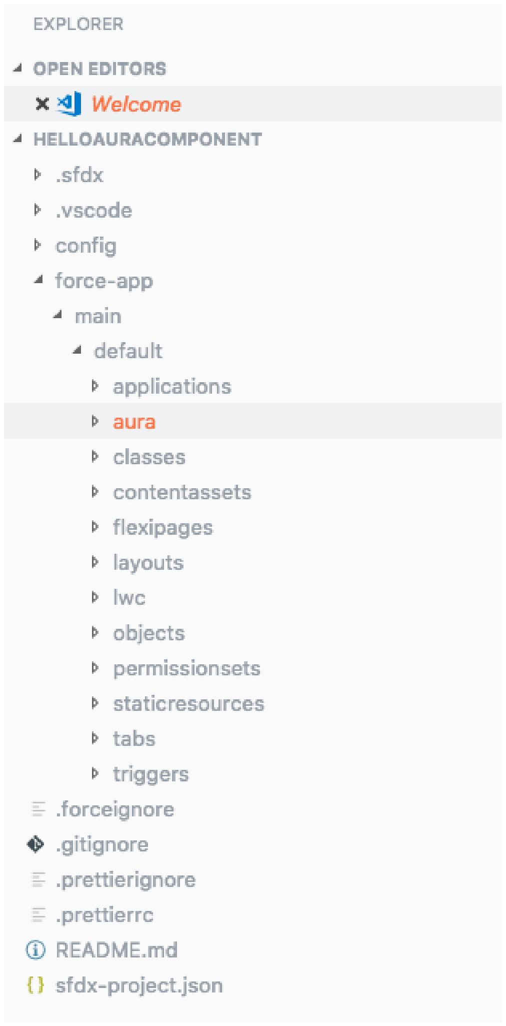
Use your favorite code editor with Salesforce CLI. We recommend using Visual Studio Code because its Salesforce Extension Pack provides powerful features for working with Salesforce CLI, the Lightning Component framework, Apex, and Visualforce.

If you choose to work with Visual Studio Code, install it and the Salesforce Extension Pack.

- Visual Studio Code (VS Code)
- Salesforce Extension Pack for Visual Studio Code

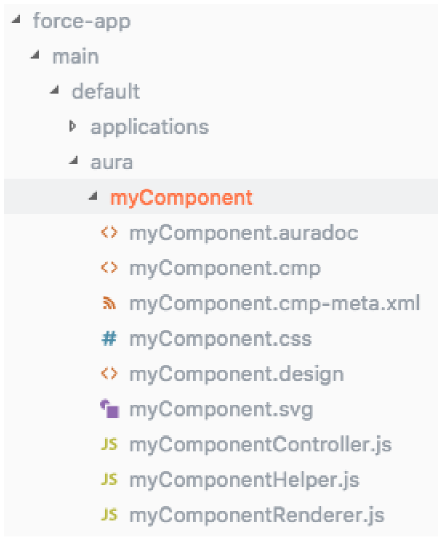
To create and deploy an Aura Component to your org:

1. Create a Salesforce DX project.
   a. In Visual Studio code, open the Command Palette by pressing **Ctrl+Shift+P** on Windows or **Cmd+Shift+P** on macOS.
   b. Type *`SFDX`* and then select **SFDX: Create Project**.
   c. Enter *`HelloAuraComponent`* and then press **Enter**. Select a folder to store the project.
   d. Click **Create Project**. You should see something like this in your Visual Studio Code workspace.



Note: The default Salesforce DX project structure facilitates moving source to and from your orgs. See Create a Salesforce DX Project.

2. Create an Aura component.
   a. Open the Command Palette and select **SFDX: Create Lightning Component**.
   b. Enter a name for your component, such as *`myComponent`*. Press **Enter**.
   c. Enter the directory for your component or press **Enter** to accept the default. The default directory is `force-app/main/default/aura`. You should see a similar directory like this.



d. Open **myComponent.cmp** and replace its content.

```
<aura:component>
    Hello World!
</aura:component>
```

3. Authenticate to your org. This step uses a Dev Hub org.

Note: You can develop Aura components in scratch orgs and non-scratch orgs. A Dev Hub org enables you to create scratch orgs. Configure an org as a Dev Hub by following the instructions at Salesforce DX Developer Guide.

a. Open the Command Palette and select **SFDX: Authorize a Dev Hub Org**. A browser window opens with a Salesforce login page.

b. Log in to your org. If prompted to allow access, click **Allow**.

After you authenticate in the browser, the CLI remembers your credentials. The success message looks like this.

```
13:40:34.679 sfdx org:login:web --alias <alias> --set-default-dev-hub
Successfully authorized username@my.org with org ID 00D1a0000000000000
13:41:48.720 sfdx org:login:web --alias <alias> --set-default-dev-hub ended with exit
 code 0
```

If the authentication fails, follow the troubleshooting guide at Salesforce CLI Setup Guide.

4. Deploy your files.

a. In the Visual Studio Code terminal, run this command.

```
sf project deploy start --source-dir force-app --target-org username@my.org
```

The success message looks like this.

```
Deployed Source
═════════════════════════════════════════════════════════════════════════════════════
| State    Name         Type                     Path
| ──────── ──────────── ──────────────────────── 
───────────────────────────────────────────────────────────────────────────────────
```

```
| Created myComponent AuraDefinitionBundle
force-app/main/default/aura/myComponent/myComponent.auradoc
| Created myComponent AuraDefinitionBundle
force-app/main/default/aura/myComponent/myComponent.cmp
| Created myComponent AuraDefinitionBundle
force-app/main/default/aura/myComponent/myComponent.cmp-meta.xml
| Created myComponent AuraDefinitionBundle
force-app/main/default/aura/myComponent/myComponent.css
| Created myComponent AuraDefinitionBundle
force-app/main/default/aura/myComponent/myComponent.design
| Created myComponent AuraDefinitionBundle
force-app/main/default/aura/myComponent/myComponent.svg
| Created myComponent AuraDefinitionBundle
force-app/main/default/aura/myComponent/myComponentController.js
| Created myComponent AuraDefinitionBundle
force-app/main/default/aura/myComponent/myComponentHelper.js
| Created myComponent AuraDefinitionBundle
force-app/main/default/aura/myComponent/myComponentRenderer.js
```

If you make changes to your component via the Developer Console in the Dev Hub org, use the `project retrieve start` command to retrieve your changes. The source you retrieve overwrites the corresponding source files in your local project.

```
sf project retrieve start --source-dir force-app --target-org username@my.org
```

SEE ALSO:

- Component Bundles
- *Salesforce DX Developer Guide:* Develop Against Any Org
- *Salesforce DX Developer Guide:* Pull Source from the Scratch Org to Your Project
- Using the Developer Console

Component Markup

Component resources contain markup and have a `.cmp` suffix. The markup can contain text or references to other components, and also declares metadata about the component.

Let's start with a simple "Hello, world!" example in a `helloWorld.cmp` component.

```
<aura:component>
    Hello, world!
</aura:component>
```

This is about as simple as a component can get. The "Hello, world!" text is wrapped in the `<aura:component>` tags, which appear at the beginning and end of every component definition.

Components can contain most HTML tags so you can use markup, such as `<div>` and `<span>`. HTML5 tags are also supported.

```
<aura:component>
    <div class="container">
        <!--Other HTML tags or components here-->
    </div>
</aura:component>
```

Note: Case sensitivity should be respected as your markup interacts with JavaScript, CSS, and Apex.

SEE ALSO:

Using the Developer Console

Component Names

Component Access Control

Component Namespace

Every component is part of a namespace, which is used to group related components together. If your organization has a namespace prefix set, use that namespace to access your components. Otherwise, use the default namespace to access your components.

Another component or application can reference a component by adding `<myNamespace:myComponent>` in its markup. For example, the `helloWorld` component is in the `docsample` namespace. Another component can reference it by adding `<docsample:helloWorld />` in its markup.

Lightning components that Salesforce provides are grouped into several namespaces, such as `aura`, `lightning`, and `force`. Components from third-party managed packages have namespaces from the providing organizations.

In your organization, you can choose to set a namespace prefix. If you do, that namespace is used for all of your Lightning components. A namespace prefix is required if you plan to offer managed packages on the AppExchange.

If you haven't set a namespace prefix for your organization, use the default namespace `c` when referencing components that you've created.

Namespaces in Code Samples

The code samples throughout this guide use the default `c` namespace. Replace `c` with your namespace if you've set a namespace prefix.

Using the Default Namespace in Organizations with No Namespace Set

If your organization hasn't set a namespace prefix, use the default namespace `c` when referencing Lightning components that you've created.

The following items must use the `c` namespace when your organization doesn't have a namespace prefix set.

- References to components that you've created
- References to events that you've defined

The following items use an implicit namespace for your organization and don't require you to specify a namespace.

- References to custom objects
- References to custom fields on standard and custom objects
- References to Apex controllers

See Namespace Usage Examples and Reference on page 26 for examples of all of the preceding items.

Using Your Organization’s Namespace

If your organization has set a namespace prefix, use that namespace to reference Lightning components, events, custom objects and fields, and other items in your Lightning markup.

The following items use your organization’s namespace when your organization has a namespace prefix set.

- References to components that you’ve created
- References to events that you’ve defined
- References to custom objects
- References to custom fields on standard and custom objects
- References to Apex controllers
- References to static resources

Note: Support for the `c` namespace in organizations that have set a namespace prefix is incomplete. The following items can use the `c` namespace if you prefer to use the shortcut, but it’s not currently a recommended practice.

- References to components that you’ve created when used in Lightning markup, but not in expressions or JavaScript
- References to events that you’ve defined when used in Lightning markup, but not in expressions or JavaScript
- References to custom objects when used in component and event `type` and `default` system attributes, but not in expressions or JavaScript

See Namespace Usage Examples and Reference on page 26 for examples of the preceding items.

Using a Namespace in or from a Managed Package

Always use the complete namespace when referencing items from a managed package, or when creating code that you intend to distribute in your own managed packages.

Another component or application can reference a component by adding `<pkgNamespace:pkgComponent>` in its markup. For example, let’s look at a package that contains the `helloWorld` component in the `docsample` namespace. Another component can reference the component from the package by adding `<docsample:helloWorld />` in its markup.

SEE ALSO:

Namespace Usage Examples and Reference

Creating a Namespace in Your Organization

Create a namespace for your organization by registering a namespace prefix.

If you’re not creating managed packages for distribution then registering a namespace prefix isn’t required, but it’s a best practice for all but the smallest organizations.

Your namespace must:

- Begin with a letter
- Contain one to 15 alphanumeric characters
- Not contain two consecutive underscores

For example, `myNp123` and `my_np` are valid namespaces, but `123Company` and `my__np` aren’t.

To register a namespace:

1. From Setup, enter `Package Manager` in the Quick Find box and select **Package Manager**.
2. In the Namespace Settings panel, click **Edit**.

   Note: After you've configured your namespace settings, this button is hidden.

3. Enter the namespace you want to register.
4. Click **Check Availability** to determine if the namespace is already in use.
5. If the namespace prefix that you entered isn't available, repeat the previous two steps.
6. Click **Review**.
7. Click **Save**.

Namespace Usage Examples and Reference

This topic provides examples of referencing components, objects, fields, and so on, in Aura components code.

Examples are provided for the following.

- Components, events, and interfaces in your organization
- Custom objects in your organization
- Custom fields on standard and custom objects in your organization
- Server-side Apex controllers in your organization
- Dynamic creation of components in JavaScript
- Static resources in your organization

Organizations with No Namespace Prefix Set

The following illustrates references to elements in your organization when your organization doesn't have a namespace prefix set. References use the default namespace, `c`, where necessary.

Referenced Item	Example
Component used in markup	`<c:myComponent />`

Component used in a system attribute	`<aura:component extends="c:myComponent">`
	`<aura:component implements="c:myInterface">`

Apex controller	`<aura:component controller="ExpenseController">`
Custom object in attribute data type	`<aura:attribute name="expense" type="Expense__c" />`

Custom object or custom field in attribute defaults	`<aura:attribute name="newExpense" type="Expense__c"`
	`    default="{ 'sobjectType': 'Expense__c',`
	`                'Name': '',`
	`                'Amount__c': 0,`
	`                …`
	`    }" />`

Referenced Item	Example
Custom field in an expression	`<lightning:inputNumber type="number" value="{!v.newExpense.Amount__c}" label=… />`
Custom field in a JavaScript function	`updateTotal: function(component) { … for(var i = 0 ; i < expenses.length ; i++){ var exp = expenses[i]; total += exp.Amount__c; } … }`
Component created dynamically in a JavaScript function	`var myCmp = $A.createComponent("c:myComponent", {}, function(myCmp) { } );`
Interface comparison in a JavaScript function	`aCmp.isInstanceOf("c:myInterface")`
Event registration	`<aura:registerEvent type="c:updateExpenseItem" name=… />`
Event handler	`<aura:handler event="c:updateExpenseItem" action=… />`
Explicit dependency	`<aura:dependency resource="markup://c:myComponent" />`
Application event in a JavaScript function	`var updateEvent = $A.get("e.c:updateExpenseItem");`
Static resources	`<ltng:require scripts="{!$Resource.resourceName}" styles="{!$Resource.resourceName}" />`

Organizations with a Namespace Prefix

The following illustrates references to elements in your organization when your organization has set a namespace prefix. References use an example namespace `yournamespace`.

Referenced Item	Example
Component used in markup	`<yournamespace:myComponent />`
Component used in a system attribute	`<aura:component extends="yournamespace:myComponent">` `<aura:component implements="yournamespace:myInterface">`
Apex controller	`<aura:component controller="yournamespace.ExpenseController">`
Custom object in attribute data type	`<aura:attribute name="expenses" type="yournamespace__Expense__c[]" />`

Referenced Item	Example
Custom object or custom field in attribute defaults	`<aura:attribute name="newExpense"` `type="yournamespace__Expense__c"` `default="{ 'sobjectType': 'yournamespace__Expense__c',` `'Name': '',` `'yournamespace__Amount__c': 0,` `…` `}" />`
Custom field in an expression	`<lightning:input type="number"` `value="{!v.newExpense.yournamespace__Amount__c}" label=… />`
Custom field in a JavaScript function	`updateTotal: function(component) {` `…` `for(var i = 0 ; i < expenses.length ; i++){` `var exp = expenses[i];` `total += exp.yournamespace__Amount__c;` `}` `…` `}`
Component created dynamically in a JavaScript function	`var myCmp = $A.createComponent("yournamespace:myComponent",` `{},` `function(myCmp) { }` `);`
Interface comparison in a JavaScript function	`aCmp.isInstanceOf("yournamespace:myInterface")`
Event registration	`<aura:registerEvent type="yournamespace:updateExpenseItem"` `name=… />`
Event handler	`<aura:handler event="yournamespace:updateExpenseItem"` `action=… />`
Explicit dependency	`<aura:dependency resource="markup://yournamespace:myComponent"` `/>`
Application event in a JavaScript function	`var updateEvent = $A.get("e.yournamespace:updateExpenseItem");`
Static resources	`<ltng:require` `scripts="{!$Resource.yournamespace__resourceName}"` `styles="{!$Resource.yournamespace__resourceName}" />`

Component Bundles

A component bundle contains a component or an app and all its related resources.

Resource	Resource Name	Usage	See Also
Component or Application	`sample.cmp` or `sample.app`	The only required resource in a bundle. Contains markup for the component or app. Each bundle contains only one component or app resource.	Creating Components on page 16 aura:application on page 465
CSS Styles	`sample.css`	Contains styles for the component.	CSS in Components on page 33
Controller	`sampleController.js`	Contains client-side controller methods to handle events in the component.	Handling Events with Client-Side Controllers on page 262
Design	`sample.design`	File required for components used in Lightning App Builder, Lightning pages, Experience Builder, or Flow Builder.	Aura Component Bundle Design Resources
Documentation	`sample.auradoc`	A description, sample code, and one or multiple references to example components	Writing Documentation for the Component Library
Renderer	`sampleRenderer.js`	Client-side renderer to override default rendering for a component.	Create a Custom Renderer on page 355
Helper	`sampleHelper.js`	JavaScript functions that can be called from any JavaScript code in a component's bundle	Sharing JavaScript Code in a Component Bundle on page 338
SVG File	sample.svg	Custom icon resource for components used in the Lightning App Builder or Experience Builder.	Configure Components for Lightning Pages and the Lightning App Builder on page 179

All resources in the component bundle follow the naming convention and are auto-wired. For example, a controller `<componentName>Controller.js` is auto-wired to its component, which means that you can use the controller within the scope of that component.

File-Size Limits

Each resource, such as a `cmp`, `css`, or `js` file, in an Aura component bundle has a maximum file size of 1,000,000 bytes. However, for performance reasons, we recommend that you don't exceed a maximum file size of 128 KB (131,072 bytes).

Component IDs

A component has two types of IDs: a local ID and a global ID. You can retrieve a component using its local ID in your JavaScript code. A global ID can be useful to differentiate between multiple instances of a component or for debugging purposes.

Local IDs

A local ID is an ID that is only scoped to the component. A local ID is often unique but it's not required to be unique.

Create a local ID by using the `aura:id` attribute. For example:

```
<lightning:button aura:id="button1" label="button1"/>
```

Note: `aura:id` doesn't support expressions. You can only assign literal string values to `aura:id`.

Find the button component by calling `cmp.find("button1")` in your client-side controller, where `cmp` is a reference to the component containing the button.

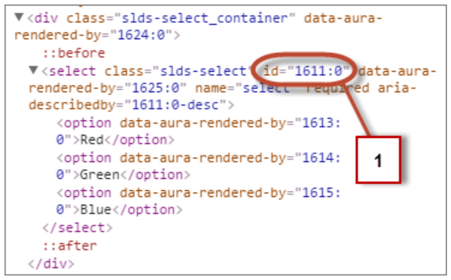
`find()` returns different types depending on the result.

- If the local ID is unique, `find()` returns the component.
- If there are multiple components with the same local ID, `find()` returns an array of the components.
- If there is no matching local ID, `find()` returns `undefined`.

To find the local ID for a component in JavaScript, use `cmp.getLocalId()`.

Global IDs

Every component has a unique `globalId`, which is the generated runtime-unique ID of the component instance. A global ID (1) is not guaranteed to be the same beyond the lifetime of a component, so it should never be relied on. A global ID can be useful to differentiate between multiple instances of a component or for debugging purposes.



```
▼<div class="slds-select_container" data-aura-
rendered-by="1624:0">
    ::before
  ▼<select class="slds-select" id="1611:0" data-aura-
  rendered-by="1625:0" name="select" required aria-
  describedby="1611:0-desc">
      <option data-aura-rendered-by="1613:
      0">Red</option>
      <option data-aura-rendered-by="1614:
      0">Green</option>
      <option data-aura-rendered-by="1615:
      0">Blue</option>
    </select>
    ::after
  </div>
```

To create a unique ID for an HTML element, you can use the `globalId` as a prefix or suffix for your element. For example:

```
<div id="{!globalId + '_footer'}"></div>
```

In your browser's developer console, retrieve the element using `document.getElementById("<globalId>_footer")`, where `<globalId>` is the generated runtime-unique ID.

To retrieve a component's global ID in JavaScript, use the `getGlobalId()` function.

```
var globalId = cmp.getGlobalId();
```

SEE ALSO:

Finding Components by ID

Which Button Was Pressed?

HTML in Components

An HTML tag is treated as a first-class component by the framework. Each HTML tag is translated into an `<aura:html>` component, allowing it to enjoy the same rights and privileges as any other component.

For example, the framework automatically converts a standard HTML `<div>` tag to this component:

```
<aura:html tag="div" />
```

You can add HTML markup in components. Note that you must use strict XHTML. For example, use `<br/>` instead of `<br>`. You can also use HTML attributes and DOM events, such as `onclick`.

Warning: Some tags, like `<applet>` and `<font>`, aren't supported.

Unescaping HTML

To output pre-formatted HTML, use `aura:unescapedHTML`. For example, this is useful if you want to display HTML that is generated on the server and add it to the DOM. You must escape any HTML if necessary or your app might be exposed to security vulnerabilities.

You can pass in values from an expression, such as in `<aura:unescapedHtml value="{!v.note.body}"/>`.

`{!`***`expression`***`}` is the framework's expression syntax. For more information, see Using Expressions on page 46.

IN THIS SECTION:

Supported HTML Tags
The framework supports most HTML tags, including the majority of HTML5 tags.

SEE ALSO:

Supported HTML Tags
CSS in Components

Supported HTML Tags

The framework supports most HTML tags, including the majority of HTML5 tags.

We recommend that you use components in preference to HTML tags. For example, use `lightning:button` instead of `<button>`.

Components are designed with accessibility in mind so users with disabilities or those who use assistive technologies can also use your app. When you start building more complex components, the reusable out-of-the-box components can simplify your job by handling some of the plumbing that you would otherwise have to create yourself. Also, these components are secure and optimized for performance.

Note that you must use strict XHTML. For example, use `<br/>` instead of `<br>`.

Some HTML tags are unsafe or unnecessary. The framework doesn't support these tags.

The `HtmlTag` enum in this Aura file lists the supported HTML tags. Any tag followed by `(false)` is not supported. For example, `applet(false)` means the `applet` tag isn't supported.

IN THIS SECTION:

Anchor Tag: <a>

Don’t hard code or dynamically generate Salesforce URLs in the `href` attribute of an `<a>` tag. Use events, such as `force:navigateToSObject` or `force:navigateToURL`, instead.

SEE ALSO:

Supporting Accessibility

Anchor Tag: `<a>`

Don’t hard code or dynamically generate Salesforce URLs in the `href` attribute of an `<a>` tag. Use events, such as `force:navigateToSObject` or `force:navigateToURL`, instead.

Avoid the `href` Attribute

Using the `href` attribute of an `<a>` tag leads to inconsistent behavior in different apps and shouldn’t be relied on. For example, don’t use this markup to link to a record:

```
<a href="/XXXXXXXXXXXXXXXXXX">Salesforce record ID (DON'T DO THIS)</a>
```

If you use `#` in the `href` attribute, a secondary issue occurs. The hash mark (#) is a URL fragment identifier and is often used in Web development for navigation within a page. Avoid `#` in the `href` attribute of anchor tags in Lightning components as it can cause unexpected navigation changes, especially in the Salesforce mobile app. That’s another reason not to use `href`.

Use the Navigation Service

Use the navigation service for consistent linking behavior across Lightning Experience, the Salesforce mobile app, and Experience Builder sites.

`lightning:navigation`
Navigates to a page or component.

`lightning:isUrlAddressable`
Enable a component to be navigated directly via a URL.

We recommend replacing these navigation events with the navigation service.

`force:navigateToList`
Navigates to a list view.

`force:navigateToObjectHome`
Navigates to an object home.

`force:navigateToRelatedList`
Navigates to a related list.

`force:navigateToSObject`
Navigates to a record.

`force:navigateToURL`
Navigates to a URL.

As well as consistent behavior, using navigation events instead of `<a>` tags reduces the number of full app reloads, leading to better performance.

SEE ALSO:

Navigate Across Your Apps with Page References

CSS in Components

Style your components with CSS.

Add CSS to a component bundle by clicking the **STYLE** button in the Developer Console sidebar.

Note: You can't add a `<style>` tag in component markup or when you dynamically create a component in JavaScript code. This restriction ensures better component encapsulation and prevents component styling interfering with the styling of another component. The `<style>` tag restriction applies to components with API version 42.0 or later.

For external CSS resources, see Styling Apps on page 308.

All top-level elements in a component have a special `THIS` CSS class added to them. This, effectively, adds namespacing to CSS and helps prevent one component's CSS from overriding another component's styling. The framework throws an error if a CSS file doesn't follow this convention.

Let's look at a sample `helloHTML.cmp` component. The CSS is in `helloHTML.css`.

Component source

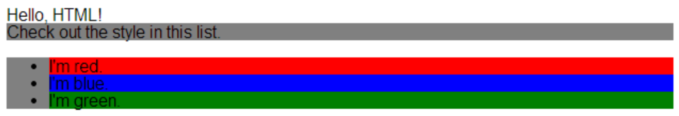
```
<aura:component>
  <div class="white">
    Hello, HTML!
  </div>

  <h2>Check out the style in this list.</h2>

  <ul>
    <li class="red">I'm red.</li>
    <li class="blue">I'm blue.</li>
    <li class="green">I'm green.</li>
  </ul>
</aura:component>
```

CSS source

```
.THIS {
    background-color: grey;
}

.THIS.white {
    background-color: white;
}

.THIS .red {
    background-color: red;
}
```

```
.THIS .blue {
    background-color: blue;
}

.THIS .green {
    background-color: green;
}
```

Output



Hello, HTML!
Check out the style in this list.

- I'm red.
- I'm blue.
- I'm green.

The top-level elements, `h2` and `ul`, match the `THIS` class and render with a grey background. Top-level elements are tags wrapped by the HTML `body` tag and not by any other tags. In this example, the `li` tags are not top-level because they are nested in a `ul` tag.

The `<div class="white">` element matches the `.THIS.white` selector and renders with a white background. Note that there is no space in the selector as this rule is for top-level elements.

The `<li class="red">` element matches the `.THIS .red` selector and renders with a red background. Note that this is a descendant selector and it contains a space as the `<li>` element is not a top-level element.

SEE ALSO:

Adding and Removing Styles

HTML in Components

Component Attributes

Component attributes are like member variables on a class in Apex. They are typed fields that are set on a specific instance of a component, and can be referenced from within the component's markup using an expression syntax. Attributes enable you to make components more dynamic.

Use the `<aura:attribute>` tag to add an attribute to the component or app. Let's look at the following sample, `helloAttributes.app`:

```
<aura:application>
    <aura:attribute name="whom" type="String" default="world"/>
    Hello {!v.whom}!
</aura:application>
```

All attributes have a name and a type. Attributes may be marked as required by specifying `required="true"`, and may also specify a default value.

In this case we've got an attribute named `whom` of type String. If no value is specified, it defaults to "world".

Though not a strict requirement, `<aura:attribute>` tags are usually the first things listed in a component's markup, as it provides an easy way to read the component's shape at a glance.

Attribute Naming Rules

An attribute name must follow these naming rules:

- Must begin with a letter or an underscore
- Must contain only alphanumeric or underscore characters

Expressions

`helloAttributes.app` contains an expression, `{!v.whom}`, which is responsible for the component's dynamic output.

`{!`***`expression`***`}` is the framework's expression syntax. In this case, the expression we are evaluating is `v.whom`. The name of the attribute we defined is `whom`, while `v` is the value provider for a component's attribute set, which represents the view.

Note: Expressions are case sensitive. For example, if you have a custom field `myNamespace__Amount__c`, you must refer to it as `{!v.myObject.myNamespace__Amount__c}`.

IN THIS SECTION:

Supported aura:attribute Types

`aura:attribute` describes an attribute available on an app, interface, component, or event.

Basic Types

Function Type

An attribute of an `aura:method` can have a type corresponding to a JavaScript function so that you can pass a function into the method. An attribute of a component can’t have a type corresponding to a JavaScript function.

Object Types

Standard and Custom Object Types

Collection Types

Custom Apex Class Types

An Aura component attribute type can correspond to values held in an Apex class. An attribute type can be a custom Apex class, a List standard Apex class, or a Map standard Apex class. To use values held in other standard Apex classes, first create a custom Apex class, and then copy the needed values from instances of the standard class into your custom class.

Framework-Specific Types

SEE ALSO:

Supported aura:attribute Types

Using Expressions

Supported aura:attribute Types

`aura:attribute` describes an attribute available on an app, interface, component, or event.

Attribute Name	Type	Description
`access`	String	Indicates whether the attribute can be used outside of its own namespace. Possible values are `public` (default), and `global`, and `private`.
`name`	String	Required. The name of the attribute. For example, if you set `<aura:attribute name="isTrue" type="Boolean" />` on a component called `aura:newCmp`, you can set this attribute when you

Attribute Name	Type	Description
		instantiate the component; for example,`<aura:newCmp isTrue="false" />`.
`type`	String	Required. The type of the attribute. For a list of basic types supported, see Basic Types.
`default`	String	The default value for the attribute, which can be overwritten as needed. When setting a default value, expressions using the `$Label`, `$Locale`, and `$Browser` global value providers are supported. Alternatively, to set a dynamic default, use an `init` event. See Invoking Actions on Component Initialization on page 337.
`required`	Boolean	Determines if the attribute is required. The default is `false`.
`description`	String	A summary of the attribute and its usage.

All `<aura:attribute>` tags have name and type values. For example:

```
<aura:attribute name="whom" type="String" />
```

Note: Although type values are case insensitive, case sensitivity should be respected as your markup interacts with JavaScript, CSS, and Apex.

SEE ALSO:

Component Attributes

Basic Types

Here are the supported basic type values. Some of these types correspond to the wrapper objects for primitives in Java. Since the framework is written in Java, defaults, such as maximum size for a number, for these basic types are defined by the Java objects that they map to.

type	Example	Description
`Boolean`	`<aura:attribute name="showDetail" type="Boolean" />`	Valid values are `true` or `false`. To set a default value of `true`, add `default="true"`.
`Date`	`<aura:attribute name="startDate" type="Date" />`	A date corresponding to a calendar day in the format yyyy-mm-dd. The hh:mm:ss portion of the date isn't stored. To include time fields, use `DateTime` instead.
`DateTime`	`<aura:attribute name="lastModifiedDate" type="DateTime" />`	A date corresponding to a timestamp. It includes date and time details with millisecond precision.

type	Example	Description
`Decimal`	`<aura:attribute name="totalPrice" type="Decimal" />`	`Decimal` values can contain fractional portions (digits to the right of the decimal). Maps to java.math.BigDecimal. `Decimal` is better than `Double` for maintaining precision for floating-point calculations. It's preferable for currency fields.
`Double`	`<aura:attribute name="widthInchesFractional" type="Double" />`	`Double` values can contain fractional portions. Maps to java.lang.Double. Use `Decimal` for currency fields instead.
`Integer`	`<aura:attribute name="numRecords" type="Integer" />`	`Integer` values can contain numbers with no fractional portion. Maps to java.lang.Integer, which defines its limits, such as maximum size.
`Long`	`<aura:attribute name="numSwissBankAccount" type="Long" />`	`Long` values can contain numbers with no fractional portion. Maps to java.lang.Long, which defines its limits, such as maximum size. Use this data type when you need a range of values wider than those provided by `Integer`.
`String`	`<aura:attribute name="message" type="String" />`	A sequence of characters.

You can use arrays for each of these basic types. For example:

```
<aura:attribute name="favoriteColors" type="String[]" default="['red','green','blue']" />
```

Retrieving Data from an Apex Controller

This example retrieves a string array of favorite colors from an Apex controller.

This component is bound to the `AttributeTypes` Apex controller and retrieves the string array when the `Update` button is clicked. The colors are displayed using a `favoriteColors` attribute.

```
<aura:component controller="AttributeTypes">
    <aura:attribute name="favoriteColors" type="String[]" default="['cyan', 'yellow',
'magenta']"/>
    <aura:iteration items="{!v.favoriteColors}" var="s">
        <p>{!s}</p>
    </aura:iteration>
    <lightning:button onclick="{!c.getString}" label="Update"/>
</aura:component>
```

The Apex controller has a `getStringArray()` method that returns a `String[]`.

```
public class AttributeTypes {

 @AuraEnabled
    public static String[] getStringArray() {
        String[] arrayItems = new String[]{ 'red', 'green', 'blue' };
        return arrayItems;
    }

}
```

The component's client-side controller retrieves the string array from the Apex controller by calling `getStringArray()`. The controller then sets the result in the `favoriteColors` attribute, which is refreshed in the UI.

```
({
    getString : function(component, event) {
    var action = component.get("c.getStringArray");
     action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                var stringItems = response.getReturnValue();
                component.set("v.favoriteColors", stringItems);
            }
        });
        $A.enqueueAction(action);
    }
})
```

To retrieve data from an object that's returned by an Apex controller, create an attribute with a type corresponding to a standard or custom object.

```
<aura:attribute name="accounts" type="Account[]"/>
```

You can access a field on the object using the `{!account.fieldName}` syntax. For more information, see Using Apex to Work with Salesforce Records.

Function Type

An attribute of an `aura:method` can have a type corresponding to a JavaScript function so that you can pass a function into the method. An attribute of a component can't have a type corresponding to a JavaScript function.

For an example of using a function type with `aura:method`, see Return Result for Asynchronous Code.

Note: Don't send attributes with `type="Function"` to the server. These attributes are intended to only be used on the client side.

The most robust way to communicate between components is to use an event. If you get an error in a component with an attribute of type `Function`, fire an event in the child component instead and handle it in the parent component.

Object Types

An attribute can have a type corresponding to an Object. For example:

```
<aura:attribute name="data" type="Object" />
```

Warning: We recommend using `type="Map"` instead of `type="Object"` to avoid some deserialization issues on the server. For example, when an attribute of `type="Object"` is serialized to the server, everything is converted to a string. Deep expressions, such as `v.data.property` can throw an exception when they are evaluated as a string on the server. Using `type="Map"` avoids these exceptions for deep expressions, and other deserialization issues.

Checking for Types

To determine a variable type, use `typeof` or a standard JavaScript method instead. The `instanceof` operator is unreliable due to the potential presence of multiple windows or frames.

SEE ALSO:

- Using Apex to Work with Salesforce Records
- Using Apex to Work with Salesforce Records

Standard and Custom Object Types

An attribute can have a type corresponding to a standard or custom object. For example, this is an attribute for a standard `Account` object:

```
<aura:attribute name="acct" type="Account" />
```

This is an attribute for an `Expense__c` custom object:

```
<aura:attribute name="expense" type="Expense__c" />
```

SEE ALSO:

- Using Apex to Work with Salesforce Records
- Using Apex to Work with Salesforce Records

Collection Types

Here are the supported collection type values.

type	Example	Description
type[] (Array)	`<aura:attribute name="colorPalette" type="String[]" default="['red', 'green', 'blue']" />`	An array of items of a defined type. Note: To set a default value, surround comma-separated values with `[]`; for example `default="['red', 'green', 'blue']"`. Setting a default value without square brackets is deprecated and can lead to unexpected behavior.
`List`	`<aura:attribute name="colorPalette" type="List" default="['red', 'green', 'blue']" />`	An ordered collection of items. Note: To set a default value, surround comma-separated values with `[]`; for example `default="['red', 'green', 'blue']"`. Setting a default value without square brackets is deprecated and can lead to unexpected behavior.

type	Example	Description
`Map`	`<aura:attribute name="sectionLabels" type="Map" default="{ a: 'label1', b: 'label2' }" />`	A collection that maps keys to values. A map can't contain duplicate keys. Each key can map to at most one value. An attribute with no default value defaults to `null` in JavaScript. If you want to set map values in JavaScript, use `default="{}"` in markup for an empty map.
`Set`	`<aura:attribute name="collection" type="Set" default="['red', 'green', 'blue']" />`	A collection that contains no duplicate elements. The order for set items is not guaranteed. For example, `"['red', 'green', 'blue']"` might be returned as `blue,green,red`. Note: To set a default value, surround comma-separated values with `[]`; for example `default="['red', 'green', 'blue']"`. Setting a default value without square brackets is deprecated and can lead to unexpected behavior.

Checking for Types

To determine a variable type, use `typeof` or a standard JavaScript method, such as `Array.isArray()`, instead. The `instanceof` operator is unreliable due to the potential presence of multiple windows or frames.

Setting List Items

There are several ways to set items in a list. To use a client-side controller, create an attribute of type List and set the items using `component.set()`.

This example retrieves a list of numbers from a client-side controller when a button is clicked.

```
<aura:attribute name="numbers" type="List"/>
<lightning:button onclick="{!c.getNumbers}" label="Display Numbers" />
<aura:iteration var="num" items="{!v.numbers}">
  {!num.value}
</aura:iteration>
```

```
/** Client-side Controller **/
({
  getNumbers: function(component, event, helper) {
    var numbers = [];
    for (var i = 0; i < 20; i++) {
      numbers.push({
        value: i
      });
    }
    component.set("v.numbers", numbers);
    }
})
```

Working with Map Items

To add a key and value pair to a map, use the syntax `myMap['myNewKey'] = myNewValue`.

```
var myMap = cmp.get("v.sectionLabels");
myMap['c'] = 'label3';
```

To retrieve a value, use `cmp.get("v.sectionLabels")['a']`.

Here's a controller with a function that adds a value, retrieves a value, and iterates over a map.

```
({
    addToMap : function(cmp, event, helper) {
        var myMap = cmp.get("v.sectionLabels");
        myMap['c'] = 'label3';
        console.log("myMap: " + JSON.stringify(myMap));

        // get map entry
        var entryA = myMap['a'];
        console.log("entryA: " + entryA);

        // iterate map
        for (var key in myMap){
            console.log("key: " + key + ", value: " + myMap[key]);
        }
    }
})
```

SEE ALSO:

Passing Data to an Apex Controller

Custom Apex Class Types

An Aura component attribute type can correspond to values held in an Apex class. An attribute type can be a custom Apex class, a List standard Apex class, or a Map standard Apex class. To use values held in other standard Apex classes, first create a custom Apex class, and then copy the needed values from instances of the standard class into your custom class.

See the List standard Apex class and the Map standard Apex class in the Apex Reference Guide.

Note: Custom Apex classes used for Aura component attributes can't be inner classes or use inheritance. Although these Apex language features can work in some situations, there are known issues, and their use is unsupported in all cases.

Supported Apex Data Types

When an instance of an Apex class is returned from a server-side action, the framework serializes the return data into JSON format. Only the values of `public` instance properties and methods annotated with `@AuraEnabled` are serialized and returned.

These Apex data types are serialized from `@AuraEnabled` properties and methods. They are supported as Aura component attributes.

- Primitive types except for BLOB

- Objects
- sObjects
- Lists and Maps if they hold elements of a supported type

Custom Apex Class Example

This example shows an Aura component called `AccountCardComponent`. When a user clicks **Show Account**, the page shows a card with the name of the account that has the specified ID.

`AccountData` is a custom Apex wrapper class that holds three strings.

```
<!-- AccountData.apxc -->
public class AccountData {
    @AuraEnabled public String accountName {get;set;}
    @AuraEnabled public String accountOwnershipType {get;set;}
    @AuraEnabled public String accountIndustry {get;set;}
}
```

The Apex server-side controller creates an `AccountData` object, and then returns the object to the JavaScript client-side controller.

```
<!-- AccountCardApexController.apxc -->
public class AccountCardApexController {
    @AuraEnabled
    public static AccountData account() {
        Account myAccount = [SELECT Name FROM Account WHERE Id='001Z5000001QaPTIA0'];
        AccountData account = new AccountData();
        account.accountName = myAccount.Name;
        return account;
    }
}
```

The JavaScript client-side controller retrieves the object from the Apex server-side controller by calling `getAccount()`. The controller then sets the result in the component's `account` attribute, which is refreshed in the UI.

```
<!-- AccountCardController.js -->
({
    getAccount : function(cmp) {
        var action = cmp.get("c.account");
        action.setCallback(this, function(response) {
            var state = response.getState();
            if(cmp.isValid() && state === "SUCCESS") {
                var accountResponse = response.getReturnValue();
                cmp.set("v.account", accountResponse);
            }
        });
        $A.enqueueAction(action);
    }
})
```

The Aura component markup contains an `<aura:attribute>` tag with `name="account"`. The attribute `type` is set to `"AccountData"`.

```
<!-- AccountCardComponent.cmp -->
<aura:component implements="flexipage:availableForAllPageTypes" access="global"
    controller="AccountCardApexController">
    <aura:attribute name="account" type="AccountData" />
    <lightning:button onclick="{!c.getAccount}" label="Show Account"/>
    <div class="slds-p-bottom_medium" />
    <aura:if isTrue="{!v.account}">
        <lightning:card title="{!v.account.accountName}">
        </lightning:card>
    </aura:if>
</aura:component>
```

List of Custom Apex Objects Example

If an attribute can contain more than one element, use a list.

This example shows an Aura component called `AccountCardsComponent` that uses the same `AccountData` custom Apex class as the previous example. When a user clicks **Show Accounts**, the page displays a list of cards with the account's name, industry, and ownership type.

The Apex server-side controller creates an list of `AccountData` objects, and then returns the list to the JavaScript client-side controller.

```
<!-- AccountCardsApexController.apxc -->
public class AccountCardsApexController {
    @AuraEnabled
    public static List<AccountData> accounts() {
    List<Account> MyAccounts = [SELECT Name, Industry, Ownership
        FROM Account LIMIT 50];
        List<AccountData> accounts = new List<AccountData>();
        for(Account acc:MyAccounts) {
            AccountData account = new AccountData();
            account.accountName = acc.Name;
            account.accountIndustry = acc.Industry;
            account.accountOwnershipType = acc.Ownership;
            accounts.add(account);
        }
        return accounts;
    }
}
```

The JavaScript client-side controller retrieves the list from the Apex server-side controller by calling `getAccounts()`. The controller then sets the result in the component's `accounts` attribute, which is refreshed in the UI.

```
<!-- AccountCardsController.js -->
({
    getAccounts : function(cmp) {
        var action = cmp.get("c.accounts");
        action.setCallback(this, function(response) {
            var state = response.getState();
            if(cmp.isValid() && state === "SUCCESS") {
                var accountsResponse = response.getReturnValue();
                cmp.set("v.accounts", accountsResponse);
            }
        });
        $A.enqueueAction(action);
    }
})
```

The Aura component markup contains an `<aura:attribute>` tag with `name="accounts"`. To set `type="AccountData[]"`, array syntax is used because the attribute holds multiple `AccountData` objects.

```
<!-- AccountCardsComponent.cmp -->
<aura:component implements="flexipage:availableForAllPageTypes" access="global"
    controller="AccountCardsApexController">
    <aura:attribute name="accounts" type="AccountData[]" />
    <lightning:button onclick="{!c.getAccounts}" label="Show Accounts" />
    <div class="slds-p-bottom_medium" />
    <aura:iteration var="acc" items="{!v.accounts}">
        <lightning:card title="{!acc.accountName}">
            <p class="slds-p-horizontal_small">
                Industry: {!acc.accountIndustry} Type: {!acc.accountOwnershipType}
            </p>
        </lightning:card>
    </aura:iteration>
</aura:component>
```

SEE ALSO:

- Returning Data from an Apex Server-Side Controller
- AuraEnabled Annotation
- Using Apex to Work with Salesforce Records
- Returning Data from an Apex Server-Side Controller
- AuraEnabled Annotation
- Using Apex to Work with Salesforce Records
- *Apex Developer Guide*: Data Types

Framework-Specific Types

Here are the supported type values that are specific to the framework.

type	Example	Description
`Aura.Component`	N/A	A single component. We recommend using `Aura.Component[]` instead.
`Aura.Component[]`	`<aura:attribute name="detail" type="Aura.Component[]"/>` To set a default value for `type="Aura.Component[]"`, put the default markup in the body of `aura:attribute`. For example: `<aura:component> <aura:attribute name="detail" type="Aura.Component[]"> <p>default`	Use this type to set blocks of markup. An attribute of type `Aura.Component[]` is called a facet.

type	Example	Description

	`paragraph1</p>`
	`    </aura:attribute>`
	`    Default value is:`
	`{!v.detail}`
	`</aura:component>`	

`Aura.Action`	`<aura:attribute name="onclick" type="Aura.Action"/>`	Use this type to pass an action to a component. See Using the Aura.Action Attribute Type.

IN THIS SECTION:

Using the Aura.Action Attribute Type

An `Aura.Action` is a reference to an action in the framework. If a child component has an `Aura.Action` attribute, a parent component can pass in an action handler when it instantiates the child component in its markup. This pattern is a shortcut to pass a controller action from a parent component to a child component that it contains, and is used for `on*` handlers, such as `onclick`.

SEE ALSO:

Component Body

Component Facets

Component Body

Component Facets

Using the Aura.Action Attribute Type

An `Aura.Action` is a reference to an action in the framework. If a child component has an `Aura.Action` attribute, a parent component can pass in an action handler when it instantiates the child component in its markup. This pattern is a shortcut to pass a controller action from a parent component to a child component that it contains, and is used for `on*` handlers, such as `onclick`.

Warning: Although `Aura.Action` works for passing an action handler to a child component, we recommend registering an event in the child component and firing the event in the child's controller instead. Then, handle the event in the parent component. The event approach requires a few extra steps in creating or choosing an event and firing it but events are the standard way to communicate between components.

`Aura.Action` shouldn't be used for other use cases. Here are some known limitations of `Aura.Action`.

- Don't use `cmp.set()` in JavaScript code to reset an attribute of `type="Aura.Action"` after it's previously been set. Doing so generates an error.

```
Unable to set value for key 'c.passedAction'. Value provider does not implement
'set(key, value)'. : false
```

- Don't use `$A.enqueueAction()` in the child component to enqueue the action passed to the `Aura.Action` attribute.

Example

This example demonstrates how to pass an action handler from a parent component to a child component.

Here's the child component with the `Aura.Action` attribute. The `onclick` handler for the button uses the value of the `onclick` attribute, which has type of `Aura.Action`.

```
<!-- child.cmp -->
<aura:component>
    <aura:attribute name="onclick" type="Aura.Action"/>

    <p>Child component with Aura.Action attribute</p>
    <lightning:button label="Execute the passed action" onclick="{!v.onclick}"/>
</aura:component>
```

Here's the parent component that contains the child component in its markup.

```
<!-- parent.cmp -->
<aura:component>
    <p>Parent component passes handler action to c:child</p>
    <c:child onclick="{!c.parentAction}"/>
</aura:component>
```

When you click the button in `c:child`, the `parentAction` action in the controller of `c:parent` is executed.

Instead of an `Aura.Action` attribute, you could use `<aura:registerEvent>` to register an `onclick` event in the child component. You'd have to define the event and create an action in the child's controller to fire the event. This event-based approach requires a few extra steps but it's more in line with standard practices for communicating between components.

SEE ALSO:

Framework-Specific Types

Handling Events with Client-Side Controllers

Framework-Specific Types

Handling Events with Client-Side Controllers

Using Expressions

Expressions allow you to make calculations and access property values and other data within component markup. Use expressions for dynamic output or passing values into components by assigning them to attributes.

An expression is any set of literal values, variables, sub-expressions, or operators that can be resolved to a single value. Method calls are not allowed in expressions.

The expression syntax is: `{!`***`expression`***`}`

`expression` is a placeholder for the expression.

Anything inside the `{! }` delimiters is evaluated and dynamically replaced when the component is rendered or when the value is used by the component. Whitespace is ignored.

Note: If you're familiar with other languages, you may be tempted to read the `!` as the "bang" operator, which negates boolean values in many programming languages. In the Aura Components programming model, `{!` is simply the delimiter used to begin an expression.

If you're familiar with Visualforce, this syntax will look familiar.

Here's an example in component markup.

```
{!v.firstName}
```

In this expression, `v` represents the view, which is the set of component attributes, and `firstName` is an attribute of the component. The expression outputs the `firstName` attribute value for the component.

The resulting value of an expression can be a primitive, such as an integer, string, or boolean. It can also be a JavaScript object, a component or collection, a controller method such as an action method, and other useful results.

There is a second expression syntax: `{#`***`expression`***`}`. For more details on the difference between the two forms of expression syntax, see Data Binding Between Components.

Identifiers in an expression, such as attribute names accessed through the view, controller values, or labels, must start with a letter or underscore. They can also contain numbers or hyphens after the first character. For example, `{!v.2count}` is not valid, but `{!v.count}` is.

Important: Only use the `{! }` syntax in markup in `.app` or `.cmp` files. In JavaScript, use string syntax to evaluate an expression. For example:

```
var theLabel = cmp.get("v.label");
```

If you want to escape `{!`, use this syntax:

```
<aura:text value="{!"/>
```

This renders `{!` in plain text because the `aura:text` component never interprets `{!` as the start of an expression.

IN THIS SECTION:

Dynamic Output in Expressions

The simplest way to use expressions is to output dynamic values.

Values used in the expression can be from component attributes, literal values, booleans, and so on. For example:

```
{!v.desc}
```

In this expression, `v` represents the view, which is the set of component attributes, and `desc` is an attribute of the component. The expression is simply outputting the `desc` attribute value for the component that contains this markup.

If you're including literal values in expressions, enclose text values within single quotes, such as `{!'Some text'}`.

Include numbers without quotes, for example, `{!123}`.

For booleans, use `{!true}` for `true` and `{!false}` for `false`.

SEE ALSO:

Component Attributes

Value Providers

Conditional Expressions

Here are examples of conditional expressions using the ternary operator and the `<aura:if>` tag.

Ternary Operator

This expression uses the ternary operator to conditionally output one of two values dependent on a condition.

```
<a class="{!v.location == '/active' ? 'selected' : ''}" href="#/active">Active</a>
```

The `{!v.location == '/active' ? 'selected' : ''}` expression conditionally sets the `class` attribute of an HTML `<a>` tag, by checking whether the `location` attribute is set to `/active`. If true, the expression sets `class` to `selected`.

Using `<aura:if>` for Conditional Markup

This snippet of markup uses the `<aura:if>` tag to conditionally display an edit button.

```
<aura:attribute name="edit" type="Boolean" default="true"/>
<aura:if isTrue="{!v.edit}">
    <lightning:button label="Edit"/>
    <aura:set attribute="else">
        You can’t edit this.
    </aura:set>
</aura:if>
```

If the `edit` attribute is set to `true`, `lightning:button` displays. Otherwise, the text in the `else` attribute displays.

SEE ALSO:

Best Practices for Conditional Markup

Data Binding Between Components

When you add a component in markup, you can use an expression to initialize attribute values in the component based on attribute values of the container component. There are two forms of expression syntax, which exhibit different behaviors for data binding between the components.

This concept is a little tricky, but it will make more sense when we look at an example. Consider a `c:parent` component that has a `parentAttr` attribute. `c:parent` contains a `c:child` component with a `childAttr` attribute that’s initialized to the value of the `parentAttr` attribute. We're passing the `parentAttr` attribute value from `c:parent` into the `c:child` component, which results in a data binding, also known as a value binding, between the two components.

```
<!--c:parent-->
<aura:component>
    <aura:attribute name="parentAttr" type="String" default="parent attribute"/>

    <!-- Instantiate the child component -->
    <c:child childAttr="{!v.parentAttr}" />
</aura:component>
```

`{!v.parentAttr}` is a bound expression. Any change to the value of the `childAttr` attribute in `c:child` also affects the `parentAttr` attribute in `c:parent` and vice versa.

Now, let's change the markup from:

```
<c:child childAttr="{!v.parentAttr}" />
```

to:

```
<c:child childAttr="{#v.parentAttr}" />
```

`{#v.parentAttr}` is an unbound expression. Any change to the value of the `childAttr` attribute in `c:child` doesn't affect the `parentAttr` attribute in `c:parent` and vice versa.

Here's a summary of the differences between the forms of expression syntax.

`{#expression}` (Unbound Expressions)

Data updates behave as you would expect in JavaScript. Primitives, such as `String`, are passed by value, and data updates for the expression in the parent and child are decoupled.

Objects, such as `Array` or `Map`, are passed by reference, so changes to the data in the child propagate to the parent. However, change handlers in the parent aren't notified. The same behavior applies for changes in the parent propagating to the child.

`{!expression}` (Bound Expressions)

Data updates in either component are reflected through bidirectional data binding in both components. Similarly, change handlers are triggered in both the parent and child components.

Tip: Bi-directional data binding is expensive for performance and it can create hard-to-debug errors due to the propagation of data changes through nested components. We recommend using the `{#expression}` syntax instead when you pass an expression from a parent component to a child component unless you require bi-directional data binding.

Unbound Expressions

Let's look at another example of a `c:parentExpr` component that contains another component, `c:childExpr`.

Here is the markup for `c:childExpr`.

```
<!--c:childExpr-->
<aura:component>
    <aura:attribute name="childAttr" type="String" />

    <p>childExpr childAttr: {!v.childAttr}</p>
    <p><lightning:button label="Update childAttr"
          onclick="{!c.updateChildAttr}"/></p>
</aura:component>
```

Here is the markup for `c:parentExpr`.

```
<!--c:parentExpr-->
<aura:component>
    <aura:attribute name="parentAttr" type="String" default="parent attribute"/>

    <!-- Instantiate the child component -->
    <c:childExpr childAttr="{#v.parentAttr}" />

    <p>parentExpr parentAttr: {!v.parentAttr}</p>
    <p><lightning:button label="Update parentAttr"
          onclick="{!c.updateParentAttr}"/></p>
</aura:component>
```

The `c:parentExpr` component uses an unbound expression to set an attribute in the `c:childExpr` component.

```
<c:childExpr childAttr="{#v.parentAttr}" />
```

When we instantiate `childExpr`, we set the `childAttr` attribute to the value of the `parentAttr` attribute in `c:parentExpr`. Since the `{#v.parentAttr}` syntax is used, the `v.parentAttr` expression is not bound to the value of the `childAttr` attribute.

The `c:exprApp` application is a wrapper around `c:parentExpr`.

```
<!--c:exprApp-->
<aura:application >
    <c:parentExpr />
</aura:application>
```

In the Developer Console, click **Preview** in the sidebar for `c:exprApp` to view the app in your browser.

Both `parentAttr` and `childAttr` are set to "parent attribute", which is the default value of `parentAttr`.

Now, let's create a client-side controller for `c:childExpr` so that we can dynamically update the component. Here is the source for `childExprController.js`.

```
/* childExprController.js */
({
    updateChildAttr: function(cmp) {
        cmp.set("v.childAttr", "updated child attribute");
    }
})
```

In the Developer Console, click **Update Preview** for `c:exprApp`.

Press the **Update childAttr** button. This updates `childAttr` to "updated child attribute". The value of `parentAttr` is unchanged since we used an unbound expression.

```
<c:childExpr childAttr="{#v.parentAttr}" />
```

Let's add a client-side controller for `c:parentExpr`. Here is the source for `parentExprController.js`.

```
/* parentExprController.js */
({
    updateParentAttr: function(cmp) {
        cmp.set("v.parentAttr", "updated parent attribute");
    }
})
```

In the Developer Console, click **Update Preview** for `c:exprApp`.

Press the **Update parentAttr** button. This time, `parentAttr` is set to "updated parent attribute" while `childAttr` is unchanged due to the unbound expression.

Warning: Don't use a component's `init` event and client-side controller to initialize an attribute that is used in an unbound expression. The attribute will not be initialized. Use a bound expression instead. For more information on a component's `init` event, see Invoking Actions on Component Initialization on page 337.

Alternatively, you can wrap the component in another component. When you instantiate the wrapped component in the wrapper component, initialize the attribute value instead of initializing the attribute in the wrapped component's client-side controller.

Bound Expressions

Now, let's update the code to use a bound expression instead. Change this line in `c:parentExpr`:

```
<c:childExpr childAttr="{#v.parentAttr}" />
```

to:

```
<c:childExpr childAttr="{!v.parentAttr}" />
```

In the Developer Console, click **Update Preview** for `c:exprApp`.

Press the **Update childAttr** button. This updates both `childAttr` and `parentAttr` to "updated child attribute" even though we only set `v.childAttr` in the client-side controller of `childExpr`. Both attributes were updated since we used a bound expression to set the `childAttr` attribute.

Change Handlers and Data Binding

You can configure a component to automatically invoke a change handler, which is a client-side controller action, when a value in one of the component's attributes changes.

When you use a bound expression, a change in the attribute in the parent or child component triggers the change handler in both components. When you use an unbound expression, the change is not propagated between components so the change handler is only triggered in the component that contains the changed attribute.

Let's add change handlers to our earlier example to see how they are affected by bound versus unbound expressions.

Here is the updated markup for `c:childExpr`.

```
<!--c:childExpr-->
<aura:component>
    <aura:attribute name="childAttr" type="String" />

    <aura:handler name="change" value="{!v.childAttr}" action="{!c.onChildAttrChange}"/>

    <p>childExpr childAttr: {!v.childAttr}</p>
    <p><lightning:button label="Update childAttr"
          onclick="{!c.updateChildAttr}"/></p>
</aura:component>
```

Notice the `<aura:handler>` tag with `name="change"`, which signifies a change handler. `value="{!v.childAttr}"` tells the change handler to track the `childAttr` attribute. When `childAttr` changes, the `onChildAttrChange` client-side controller action is invoked.

Here is the client-side controller for `c:childExpr`.

```
/* childExprController.js */
({
    updateChildAttr: function(cmp) {
        cmp.set("v.childAttr", "updated child attribute");
    },

    onChildAttrChange: function(cmp, evt) {
        console.log("childAttr has changed");
        console.log("old value: " + evt.getParam("oldValue"));
        console.log("current value: " + evt.getParam("value"));
    }
})
```

Here is the updated markup for `c:parentExpr` with a change handler.

```
<!--c:parentExpr-->
<aura:component>
    <aura:attribute name="parentAttr" type="String" default="parent attribute"/>

   <aura:handler name="change" value="{!v.parentAttr}" action="{!c.onParentAttrChange}"/>


    <!-- Instantiate the child component -->
    <c:childExpr childAttr="{!v.parentAttr}" />

    <p>parentExpr parentAttr: {!v.parentAttr}</p>
    <p><lightning:button label="Update parentAttr"
          onclick="{!c.updateParentAttr}"/></p>
</aura:component>
```

Here is the client-side controller for `c:parentExpr`.

```
/* parentExprController.js */
({
    updateParentAttr: function(cmp) {
        cmp.set("v.parentAttr", "updated parent attribute");
    },

    onParentAttrChange: function(cmp, evt) {
        console.log("parentAttr has changed");
        console.log("old value: " + evt.getParam("oldValue"));
        console.log("current value: " + evt.getParam("value"));
    }
})
```

In the Developer Console, click **Update Preview** for `c:exprApp`.

Open your browser's console (**More tools** > **Developer tools** in Chrome).

Press the **Update parentAttr** button. The change handlers for `c:parentExpr` and `c:childExpr` are both triggered as we're using a bound expression.

```
<c:childExpr childAttr="{!v.parentAttr}" />
```

Change `c:parentExpr` to use an unbound expression instead.

```
<c:childExpr childAttr="{#v.parentAttr}" />
```

In the Developer Console, click **Update Preview** for `c:exprApp`.

Press the **Update childAttr** button. This time, only the change handler for `c:childExpr` is triggered as we're using an unbound expression.

SEE ALSO:

Detecting Data Changes with Change Handlers

Dynamic Output in Expressions

Component Composition

Value Providers

Value providers are a way to access data. Value providers encapsulate related values together, similar to how an object encapsulates properties and methods.

The value providers for a component are `v` (view) and `c` (controller).

Value Provider	Description	See Also
`v`	A component's attribute set. This value provider enables you to access the value of a component's attribute in the component's markup.	Component Attributes
`c`	A component's controller, which enables you to wire up event handlers and actions for the component	Handling Events with Client-Side Controllers

All components have a `v` value provider, but aren't required to have a controller. Both value providers are created automatically when defined for a component.

Note: Expressions are bound to the specific component that contains them. That component is also known as the attribute value provider, and is used to resolve any expressions that are passed to attributes of its contained components.

Global Value Providers

Global value providers are global values and methods that a component can use in expressions.

Global Value Provider	Description	See Also
`globalID`	The `globalId` global value provider returns the global ID for a component. Every component has a unique `globalId`, which is the generated runtime-unique ID of the component instance.	Component IDs
`$Browser`	The `$Browser` global value provider returns information about the hardware and operating system of the browser accessing the application.	$Browser

Global Value Provider	Description	See Also
`$ContentAsset`	The `$ContentAsset` global value provider lets you reference images, style sheets, and JavaScript used as asset files in your lightning components.	$ContentAsset
`$Label`	The `$Label` global value provider enables you to access labels stored outside your code.	Using Custom Labels
`$Locale`	The `$Locale` global value provider returns information about the current user's preferred locale.	$Locale
`$Resource`	The `$Resource` global value provider lets you reference images, style sheets, and JavaScript code you've uploaded in static resources.	$Resource

Accessing Fields and Related Objects

Values in a value provider are accessed as named properties. To use a value, separate the value provider and the property name with a dot (period). For example, `v.body`. You can access value providers in markup or in JavaScript code.

When an attribute of a component is an object or other structured data (not a primitive value), access the values on that attribute using the same dot notation.

For example, `{!v.accounts.id}` accesses the id field in the accounts record.

For deeply nested objects and attributes, continue adding dots to traverse the structure and access the nested values.

SEE ALSO:

Dynamic Output in Expressions

`$Browser`

The `$Browser` global value provider returns information about the hardware and operating system of the browser accessing the application.

Attribute	Description
`formFactor`	Returns a `FormFactor` enum value based on the type of hardware the browser is running on. • `DESKTOP` for a desktop client • `PHONE` for a phone including a mobile phone with a browser and a smartphone
`isAndroid`	Indicates whether the browser is running on an Android device (`true`) or not (`false`).
`isIOS`	Due to changes made by Apple, `isIOS` is deprecated because it no longer distinguishes between iPad and MacOS desktop.
`isIPad`	Due to changes made by Apple, `isIPad` is deprecated because it no longer distinguishes between iPad and MacOS desktop.

Attribute	Description
`isIPhone`	Not available in all implementations. Indicates whether the browser is running on an iPhone (`true`) or not (`false`).
`isPhone`	Indicates whether the browser is running on a phone including a mobile phone with a browser and a smartphone (`true`), or not (`false`).
`isTablet`	Indicates whether the browser is running on a tablet with Android 2.2 or later (`true`) or not (`false`). Note: Due to changes made by Apple, `isTablet` is deprecated for iOS devices because `$Browser` no longer distinguishes between iPad and MacOS desktop.
`isWindowsPhone`	Indicates whether the browser is running on a Windows phone (`true`) or not (`false`). This attribute detects only Windows phones and doesn't detect tablets or other touch-enabled Windows 8 devices.

Example: This example shows usage of the `$Browser` global value provider.

```
<aura:component>
    {!$Browser.isTablet}
    {!$Browser.isPhone}
    {!$Browser.isAndroid}
    {!$Browser.formFactor}
</aura:component>
```

Similarly, you can check browser information in a client-side controller using `$A.get()`.

```
({
    checkBrowser: function(component) {
        var device = $A.get("$Browser.formFactor");
        alert("You are using a " + device);
    }
})
```

`$ContentAsset`

The `$ContentAsset` global value provider lets you reference images, style sheets, and JavaScript used as asset files in your Lightning components.

Reference `$ContentAsset` asset files by name instead of using cumbersome file paths or URLs. `$ContentAsset` provides sharing, versioning, and access control for all asset files, as well as options for mobile optimization and resizing of image files. You can use `$ContentAsset` in Lightning components markup and within JavaScript controller and helper code.

Using `$ContentAsset` in Component Markup

To reference a specific asset file in component markup, use ***`$ContentAsset.yourNamespace__assetName`***. Orgs without a namespace can use ***`$ContentAsset.assetDeveloperName`***. Use this syntax regardless of whether an asset is for authenticated or unauthenticated sessions. To reference a content asset within an archive, add ***`pathinarchive`*** as a parameter appended to the basic syntax: ***`$ContentAsset.yourNamespace__assetName + 'pathinarchive=images/sampleImage.jpg'`***.

Here are a few examples.

Aura component referencing an image in an archive asset file:

```
<aura:component>
   <img src="{!$ContentAsset.websiteImages + 'pathinarchive=images/logo.jpg'}" "alt="holiday
 wreath"/>
</aura:component>
```

Include CSS style sheets or JavaScript libraries in a component using the `<ltng:require>` tag.

Aura component using an asset file to style a `div` element:

Markup

```
<aura:component>
    <ltng:require styles="{!$ContentAsset.bookStyle}"/>

    <!-- "bookName" is defined in an asset file with DeveloperName of "bookStyle" -->
    <div id="bookTitle" class="bookName">
    </div>
</aura:component>
```

Aura component displays data from a `testDisplayData` JavaScript asset file:

Markup

```
<aura:component>
    <ltng:require scripts="{!$ContentAsset.testDisplayData}"
afterScriptsLoaded="{!c.displayData}"/>
...

    <aura:attribute name="TestData" type="String[]" ></aura:attribute>
        <div>
            <input type="text" id="sampleData" value ="{!v.TestData}" />
        </div>
...
</aura:component>
```

Controller

```
({
    displayData : function(component, event, helper) {
        var data = _datamap.getData();
        component.set("v.TestData", data);
    }
})
```

JavaScript (`.js`) Asset File with DeveloperName `testDisplayData`

```
window._datamap = (function() {
    var data = ["Agree", "Disagree", "Strongly Agree", "Strongly Disagree", "Not
Applicable"];
    return {
        getData: function() {
            return data.join(", ");
        }
    };
}());
```

`$Locale`

The `$Locale` global value provider returns information about the current user's preferred locale.

Attribute	Description	Sample Value
`country`	The ISO 3166 representation of the country code based on the language locale.	"US", "DE", "GB"
`currency`	The currency symbol.	"$"
`currencyCode`	The ISO 4217 representation of the currency code.	"USD"
`decimal`	The decimal separator.	"."
`firstDayOfWeek`	The first day of the week, where 1 is Sunday.	1
`grouping`	The grouping separator.	","
`isEasternNameStyle`	Specifies if a name is based on eastern style, for example, `last name first name [middle] [suffix]`.	false
`labelForToday`	The label for the Today link on the date picker.	"Today"
`language`	The language code based on the language locale.	"en", "de", "zh"
`langLocale`	The locale ID.	"en_US", "en_GB"
`nameOfMonths`	The full and short names of the calendar months	{ fullName: "January", shortName: "Jan" }
`nameOfWeekdays`	The full and short names of the calendar weeks	{ fullName: "Sunday", shortName: "SUN" }
`timezone`	The time zone ID.	"America/Los_Angeles"
`userLocaleCountry`	The country based on the current user's locale	"US"
`userLocaleLang`	The language based on the current user's locale	"en"
`variant`	Deprecated. The variation for a language dialect.	

Number and Date Formatting

Attribute	Description	Sample Value
`currencyFormat`	The currency format.	"¤#,##0.00;(¤#,##0.00)" ¤ represents the currency sign, which is replaced by the currency symbol.
`dateFormat`	The date format.	"MMM d, yyyy"
`datetimeFormat`	The date time format.	"MMM d, yyyy h:mm:ss a"
`longDateFormat`	The long date format.	"MMMM d, yyyy"

Attribute	Description	Sample Value
`numberFormat`	The number format.	"#,##0.###" # represents a digit, the comma is a placeholder for the grouping separator, and the period is a placeholder for the decimal separator. Zero (0) replaces # to represent trailing zeros.
`percentFormat`	The percentage format.	"#,##0%"
`shortDateFormat`	The short date format.	"M/d/yyyy"
`shortDatetimeFormat`	The short date time format.	"M/d/yyyy h:mm a"
`shortTimeFormat`	The short time format.	"h:mm a"
`showJapaneseCalendar`	Specifies whether to show dates in the Japanese Imperial calendar format	false
`timeFormat`	The time format.	"h:mm:ss a"
`zero`	The character for the zero digit.	"0"

Example: This example shows how to retrieve different `$Locale` attributes.

Component source

```
<aura:component>
    {!$Locale.language}
    {!$Locale.timezone}
    {!$Locale.numberFormat}
    {!$Locale.currencyFormat}
</aura:component>
```

Similarly, you can check locale information in JavaScript using `$A.get()`.

```
({
    checkDevice: function(component) {
        var locale = $A.get("$Locale.language");
        alert("You are using " + locale);
    }
})
```

SEE ALSO:

Localization

$Resource

The `$Resource` global value provider lets you reference images, style sheets, and JavaScript code you've uploaded in static resources.

Using `$Resource` lets you reference assets by name, without worrying about the gory details of URLs or file paths. You can use `$Resource` in Aura component markup and within JavaScript controller and helper code.

Using `$Resource` in Component Markup

To reference a specific resource in component markup, use `$Resource.`***`resourceName`*** within an expression. *`resourceName`* is the `Name` of the static resource. In a managed package, the resource name must include the package namespace prefix, such as `$Resource.yourNamespace__resourceName`. For a stand-alone static resource, such as an individual graphic or script, you only need the name of the resource. For example, if you uploaded `myScript.js` and set the `Name` to `myScript`, reference it as `$Resource.myScript`. To reference an item within an archive static resource, add the rest of the path to the item using string concatenation. Here are a few examples.

```
<aura:component>
    <!-- Stand-alone static resources -->
    <img src="{!$Resource.generic_profile_svg}"/>
    <img src="{!$Resource.yourNamespace__generic_profile_svg}"/>

    <!-- Asset from an archive static resource -->
    <img src="{!$Resource.yourGraphics + '/images/logo.jpg'}"/>
    <img src="{!$Resource.yourNamespace__yourGraphics + '/images/logo.jpg'}"/>
</aura:component>
```

Include CSS style sheets or JavaScript libraries into a component using the `<ltng:require>` tag. For example:

```
<aura:component>
  <ltng:require
    styles="{!$Resource.jsLibraries  + '/styles/jsMyStyles.css'}"
    scripts="{!$Resource.jsLibraries + '/jsLibOne.js'}"
    afterScriptsLoaded="{!c.scriptsLoaded}" />
</aura:component>
```

Note: Due to a quirk in the way `$Resource` is parsed in expressions, use the `join` operator to include multiple `$Resource` references in a single attribute. For example, if you have more than one JavaScript library to include into a component the `scripts` attribute should be something like the following.

```
scripts="{!join(',',
    $Resource.jsLibraries + '/jsLibOne.js',
    $Resource.jsLibraries + '/jsLibTwo.js')}"
```

Using `$Resource` in JavaScript

To obtain a reference to a static resource in JavaScript code, use `$A.get('$Resource.`***`resourceName`***`')`.

`resourceName` is the `Name` of the static resource. In a managed package, the resource name must include the package namespace prefix, such as `$Resource.yourNamespace__resourceName`. For a stand-alone static resource, such as an individual graphic or script, you only need the name of the resource. For example, if you uploaded `myScript.js` and set the `Name` to `myScript`, reference it as `$Resource.myScript`. To reference an item within an archive static resource, add the rest of the path to the item using string concatenation. For example:

```
({
    profileUrl: function(component) {
        var profUrl = $A.get('$Resource.yourGraphics') + '/images/avatar1.jpg';
        alert("Profile URL: " + profUrl);
    }
})
```

Note: Static resources referenced in JavaScript aren't automatically added to packages. If your JavaScript depends on a resource that isn't referenced in component markup, add it manually to any packages the JavaScript code is included in.

`$Resource` Considerations

Global value providers in the Aura Components programming model are, behind the scenes, implemented quite differently from global variables in Salesforce. Although `$Resource` looks like the global variable with the same name available in Visualforce, formula fields, and elsewhere, there are important differences. Don't use other documentation as a guideline for its use or behavior.

Here are two specific things to keep in mind about `$Resource` in the Aura Components programming model.

First, `$Resource` isn't available until the Aura Components programming model is loaded on the client. Some very simple components that are composed of only markup can be rendered server-side, where `$Resource` isn't available. To avoid this, when you create a new app, stub out a client-side controller to force components to be rendered on the client.

Second, if you've worked with the `$Resource` global variable, in Visualforce or elsewhere, you've also used the `URLFOR()` formula function to construct complete URLs to specific resources. There's nothing similar to `URLFOR()` in the Aura Components programming model. Instead, use simple string concatenation, as illustrated in the preceding examples.

SEE ALSO:

Salesforce Help: Static Resources

Expression Evaluation

Expressions are evaluated much the same way that expressions in JavaScript or other programming languages are evaluated.

Operators are a subset of those available in JavaScript, and evaluation order and precedence are generally the same as JavaScript. Parentheses enable you to ensure a specific evaluation order. What you may find surprising about expressions is how often they are evaluated. The framework notices when things change, and triggers re-rendering of any components that are affected. Dependencies are handled automatically. When a component is re-rendered, any expressions it uses will be re-evaluated.

Action Methods

Expressions are also used to provide action methods for user interface events: `onclick`, `onhover`, and any other component attributes beginning with "`on`".

Action methods must be assigned to attributes using an expression, for example `{!c.theAction}`. This expression assigns a reference to the controller function that handles the action.

Assigning action methods via expressions allows you to assign them conditionally, based on the state of the application or user interface. For more information, see Conditional Expressions on page 48.

```
<aura:component>
    <aura:attribute name="liked" type="Boolean" default="true"/>
    <lightning:button aura:id="likeBtn"
     label="{!(v.liked) ? 'Like It' : 'Unlike It'}"
     onclick="{!(v.liked) ? c.likeIt  : c.unlikeIt}"
    />
</aura:component>
```

This button will show "Like It" for items that have not yet been liked, and clicking it will call the `likeIt` action method. Then the component will re-render, and the opposite user interface display and method assignment will be in place. Clicking a second time will unlike the item, and so on.

Note: The example demonstrates how attributes can help you control the state of a button. To create a button that toggles between states, we recommend using the `lightning:buttonStateful` component.

Action Methods with Lightning Web Components

If you try to use an action method with a Lightning web component, it doesn't behave as expected because Lightning web components don't support expressions the same way that Aura components do. To write an action method, assign a controller action and execute logic depending on a value.

```
<aura:component>
    <aura:attribute name="liked" type="Boolean" default="true"/>
    <c:lwcButton aura:id="likeBtn"
     label="{!(v.liked) ? 'Like It' : 'Unlike It'}"
     onclick="{!c.handleLikeButtonClick}"
    />
</aura:component>
```

```
({
    handleLikeButtonClick: function (cmp) {
        if (cmp.get('v.liked')) {
            // like it logic
        } else {
            // unlike it logic
        }
    }
})
```

Expression Operators Reference

The expression language supports operators to enable you to create more complex expressions.

Arithmetic Operators

Expressions based on arithmetic operators result in numerical values.

Operator	Usage	Description
`+`	`1 + 1`	Add two numbers.
`-`	`2 - 1`	Subtract one number from the other.
`*`	`2 * 2`	Multiply two numbers.
`/`	`4 / 2`	Divide one number by the other.
`%`	`5 % 2`	Return the integer remainder of dividing the first number by the second.
`-`	`-v.exp`	Unary operator. Reverses the sign of the succeeding number. For example if the value of `expenses` is `100`, then `-expenses` is `-100`.

Numeric Literals

Literal	Usage	Description
Integer	`2`	Integers are numbers without a decimal point or exponent.

Float	`3.14`
	`-1.1e10`
		Numbers with a decimal point, or numbers with an exponent.

Null	`null`	A literal null number. Matches the explicit null value **and** numbers with an undefined value.

String Operators

Expressions based on string operators result in string values.

Operator	Usage	Description
+	`'Title: ' + v.note.title`	Concatenates two strings together.

String Literals

String literals must be enclosed in single quotation marks `'like this'`.

Literal	Usage	Description
string	`'hello world'`	Literal strings must be enclosed in single quotation marks. Double quotation marks are reserved for enclosing attribute values, and must be escaped in strings.
\<escape>	`'\n'`	Whitespace characters: • `\t` (tab) • `\n` (newline) • `\r` (carriage return) Escaped characters: • `\"` (literal ") • `\'` (literal ') • `\\` (literal \)
Unicode	`'\u####'`	A Unicode code point. The # symbols are hexadecimal digits. A Unicode literal requires four digits.
null	`null`	A literal null string. Matches the explicit null value and strings with an undefined value.

Comparison Operators

Expressions based on comparison operators result in a `true` or `false` value. For comparison purposes, numbers are treated as the same type. In all other cases, comparisons check both value and type.

Operator	Alternative	Usage	Description

`==`	`eq`	`1 == 1`
		`1 == 1.0`
		`1 eq 1`
		Note: `undefined==null` evaluates to `true`.
			Returns `true` if the operands are equal. This comparison is valid for all data types.
			Warning: Don't use the `==` operator for objects, as opposed to basic types, such as Integer or String. For example, `object1==object2` evaluates inconsistently on the client versus the server and isn't reliable.

`!=`	`ne`	`1 != 2`
		`1 != true`
		`1 != '1'`
		`null != false`
		`1 ne 2`
			Returns `true` if the operands are not equal. This comparison is valid for all data types.

`<`	`lt`	`1 < 2`
		`1 lt 2`
			Returns `true` if the first operand is numerically less than the second. You must escape the `<` operator to `&lt;` to use it in component markup. Alternatively, you can use the `lt` operator.

`>`	`gt`	`42 > 2`
		`42 gt 2`
			Returns `true` if the first operand is numerically greater than the second.

`<=`	`le`	`2 <= 42`
		`2 le 42`
			Returns `true` if the first operand is numerically less than or equal to the second. You must escape the `<=` operator to `&lt;=` to use it in component markup. Alternatively, you can use the `le` operator.

`>=`	`ge`	`42 >= 42`
		`42 ge 42`
			Returns `true` if the first operand is numerically greater than or equal to the second.

Logical Operators

Expressions based on logical operators result in a `true` or `false` value.

Operator	Usage	Description
`&amp;&amp;`	`isEnabled &amp;&amp; hasPermission`	Returns `true` if both operands are individually true. If you have more than two arguments, you can chain multiple `&amp;&amp;` operations.

Operator	Usage	Description
		This syntax is awkward in markup so we recommend the alternative of using the `and()` function when you have two arguments. For example, `and(isEnabled, hasPermission)`. The `and()` function only works with two arguments.
`||`	`hasPermission || isRequired`	Returns `true` if either operand is individually true. If you have more than two arguments, you can chain multiple `||` operations. You can alternatively use the `or()` function when you have only two arguments. The `or()` function only works with two arguments.
`!`	`!isRequired`	Unary operator. Returns `true` if the operand is false. This operator should not be confused with the `!` delimiter used to start an expression in `{!`. You can combine the expression delimiter with this negation operator to return the logical negation of a value, for example, `{!!true}` returns `false`.

Logical Literals

Logical values are never equivalent to non-logical values. That is, only `true == true`, and only `false == false`; `1 != true`, and `0 != false`, and `null != false`.

Literal	Usage	Description
true	`true`	A boolean `true` value.
false	`false`	A boolean `false` value.

Conditional Operator

There is only one conditional operator, the traditional ternary operator.

Operator	Usage	Description
`? :`	`(1 != 2) ? "Obviously" : "Black is White"`	The operand before the `?` operator is evaluated as a boolean. If true, the second operand is returned. If false, the third operand is returned.

SEE ALSO:

Expression Functions Reference

Expression Functions Reference

The expression language contains math, string, array, comparison, boolean, and conditional functions. All functions are case-sensitive.

Math Functions

The math functions perform math operations on numbers. They take numerical arguments. The Corresponding Operator column lists equivalent operators, if any.

Function	Alternative	Usage	Description	Corresponding Operator
`add`	`concat`	`add(1,2)`	Adds the first argument to the second.	+
`sub`	`subtract`	`sub(10,2)`	Subtracts the second argument from the first.	–
`mult`	`multiply`	`mult(2,10)`	Multiplies the first argument by the second.	*
`div`	`divide`	`div(4,2)`	Divides the first argument by the second.	/
`mod`	`modulus`	`mod(5,2)`	Returns the integer remainder resulting from dividing the first argument by the second.	%
`abs`		`abs(-5)`	Returns the absolute value of the argument: the same number if the argument is positive, and the number without its negative sign if the number is negative. For example, `abs(-5) is 5`.	None
`neg`	`negate`	`neg(100)`	Reverses the sign of the argument. For example, `neg(100) is -100`.	– (unary)

String Functions

Function	Alternative	Usage	Description	Corresponding Operator
`concat`	`add`	`concat('Hello ', 'world')` `add('Walk ', 'the dog')`	Concatenates the two arguments.	+

Function	Alternative	Usage	Description	Corresponding Operator
`format`		`format($Label.ns.labelName, v.myVal)` Note: This function works for arguments of type `String`, `Decimal`, `Double`, `Integer`, `Long`, `Array`, `String[]`, `List`, and `Set`.	Replaces any parameter placeholders with comma-separated attribute values.	

`join`		`join(separator, subStr1, subStr2, subStrN)`
		 `join(' ','class1', 'class2', v.class)`
			Joins the substrings adding the separator String (first argument) between each subsequent argument.	

Label Functions

Function	Usage	Description

`format`	`format($Label.np.labelName, v.attribute1 , v.attribute2)`
	 `format($Label.np.hello, v.name)`
		Outputs a label and updates it. Replaces any parameter placeholders with comma-separated attribute values. Supports ternary operators in labels and attributes.

Informational Functions

Function	Usage	Description
`length`	`myArray.length`	Returns the length of an array or a string.
`empty`	`empty(v.attributeName)` Note: This function works for arguments of type `String`, `Array`, `Object`, `List`, `Map`, or `Set`.	Returns `true` if the argument is empty. An empty argument is `undefined`, `null`, an empty array, or an empty string. An object with no properties is not considered empty. Tip: `{! !empty(v.myArray)}` evaluates faster than `{!v.myArray && v.myArray.length > 0}` so we recommend `empty()` to improve performance. The `$A.util.isEmpty()` method in JavaScript is equivalent to the `empty()` expression in markup.

Comparison Functions

Comparison functions take two number arguments and return `true` or `false` depending on the comparison result. The `eq` and `ne` functions can also take other data types for their arguments, such as strings.

Function	Usage	Description	Corresponding Operator
`equals`	`equals(1,1)`	Returns `true` if the specified arguments are equal. The arguments can be any data type.	`==` or `eq`
`notequals`	`notequals(1,2)`	Returns `true` if the specified arguments are not equal. The arguments can be any data type.	`!=` or `ne`
`lessthan`	`lessthan(1,5)`	Returns `true` if the first argument is numerically less than the second argument.	`<` or `lt`
`greaterthan`	`greaterthan(5,1)`	Returns `true` if the first argument is numerically greater than the second argument.	`>` or `gt`
`lessthanorequal`	`lessthanorequal(1,2)`	Returns `true` if the first argument is numerically less than or equal to the second argument.	`<=` or `le`
`greaterthanorequal`	`greaterthanorequal(2,1)`	Returns `true` if the first argument is numerically greather than or equal to the second argument.	`>=` or `ge`

Boolean Functions

Boolean functions operate on Boolean arguments. They are equivalent to logical operators.

Function	Usage	Description	Corresponding Operator
`and`	`and(isEnabled, hasPermission)`	Returns `true` if both arguments are true. Note: This function supports only two arguments. Any arguments after the first two are ignored.	`&amp;&amp;` This syntax is awkward in markup so we recommend using the `and()` function instead when you have two arguments. If you have more than two arguments, you can chain multiple `&amp;&amp;` operations.

Function	Usage	Description	Corresponding Operator
`or`	`or(hasPermission, hasVIPPass)`	Returns `true` if either one of the arguments is true. Note: This function supports only two arguments. Any arguments after the first two are ignored.	`||` If you have more than two arguments, you can chain multiple `||` operations.
`not`	`not(isNew)`	Returns `true` if the argument is false.	`!`

Conditional Function

Function	Usage	Description	Corresponding Operator
`if`	`if(isEnabled, 'Enabled', 'Not enabled')`	Evaluates the first argument as a boolean. If true, returns the second argument. Otherwise, returns the third argument.	`?:` (ternary)

Component Composition

Composing fine-grained components in a larger component enables you to build more interesting components and applications.

Let's see how we can fit components together. We will first create a few simple components: `c:helloHTML` and `c:helloAttributes`. Then, we'll create a wrapper component, `c:nestedComponents`, that contains the simple components.

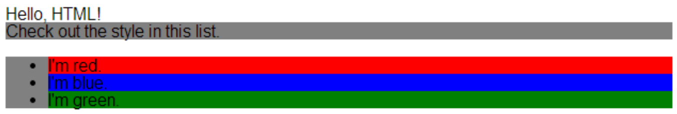
Here is the source for `helloHTML.cmp`.

```
<!--c:helloHTML-->
<aura:component>
  <div class="white">
    Hello, HTML!
  </div>

  <h2>Check out the style in this list.</h2>

  <ul>
    <li class="red">I'm red.</li>
    <li class="blue">I'm blue.</li>
    <li class="green">I'm green.</li>
  </ul>
</aura:component>
```

CSS source

```
.THIS {
    background-color: grey;
}

.THIS.white {
    background-color: white;
}

.THIS .red {
    background-color: red;
}

.THIS .blue {
    background-color: blue;
}

.THIS .green {
    background-color: green;
}
```

Output



```
Hello, HTML!
Check out the style in this list.

• I'm red.
• I'm blue.
• I'm green.
```

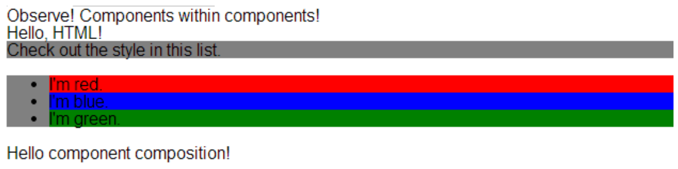
Here is the source for `helloAttributes.cmp`.

```
<!--c:helloAttributes-->
<aura:component>
    <aura:attribute name="whom" type="String" default="world"/>
    Hello {!v.whom}!
</aura:component>
```

`nestedComponents.cmp` uses composition to include other components in its markup.

```
<!--c:nestedComponents-->
<aura:component>
    Observe!  Components within components!

    <c:helloHTML/>

    <c:helloAttributes whom="component composition"/>
</aura:component>
```

Output



```
Observe! Components within components!
Hello, HTML!
Check out the style in this list.

• I'm red.
• I'm blue.
• I'm green.

Hello component composition!
```

Including an existing component is similar to including an HTML tag. Reference the component by its "descriptor", which is of the form *`namespace:component`*. `nestedComponents.cmp` references the `helloHTML.cmp` component, which lives in the `c` namespace. Hence, its descriptor is `c:helloHTML`.

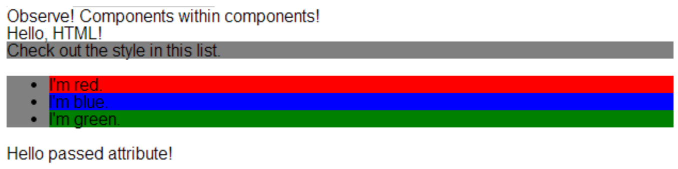
Note how `nestedComponents.cmp` also references `c:helloAttributes`. Just like adding attributes to an HTML tag, you can set attribute values in a component as part of the component tag. `nestedComponents.cmp` sets the `whom` attribute of `helloAttributes.cmp` to "component composition".

Attribute Passing

You can also pass attributes to nested components. `nestedComponents2.cmp` is similar to `nestedComponents.cmp`, except that it includes an extra `passthrough` attribute. This value is passed through as the attribute value for `c:helloAttributes`.

```
<!--c:nestedComponents2-->
<aura:component>
    <aura:attribute name="passthrough" type="String" default="passed attribute"/>
    Observe!  Components within components!

    <c:helloHTML/>

    <c:helloAttributes whom="{#v.passthrough}"/>
</aura:component>
```

Output



```
Observe! Components within components!
Hello, HTML!
Check out the style in this list.

 • I'm red.
 • I'm blue.
 • I'm green.

Hello passed attribute!
```

`helloAttributes` is now using the passed through attribute value.

Note: `{#v.passthrough}` is an unbound expression. This means that any change to the value of the `whom` attribute in `c:helloAttributes` doesn't propagate back to affect the value of the `passthrough` attribute in `c:nestedComponents2`. For more information, see Data Binding Between Components on page 48.

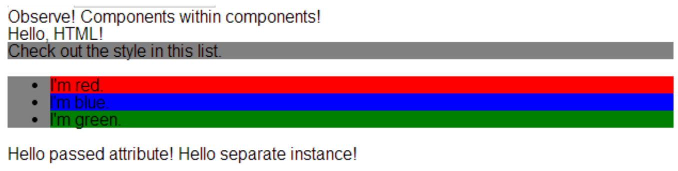
Definitions versus Instances

In object-oriented programming, there's a difference between a class and an instance of that class. Components have a similar concept. When you create a `.cmp` resource, you are providing the definition (class) of that component. When you put a component tag in a `.cmp` resource, you are creating a reference to (instance of) that component.

It shouldn't be surprising that we can add multiple instances of the same component with different attributes. `nestedComponents3.cmp` adds another instance of `c:helloAttributes` with a different attribute value. The two instances of the `c:helloAttributes` component have different values for their `whom` attribute .

```
<!--c:nestedComponents3-->
<aura:component>
    <aura:attribute name="passthrough" type="String" default="passed attribute"/>
    Observe!  Components within components!

    <c:helloHTML/>

    <c:helloAttributes whom="{#v.passthrough}"/>

    <c:helloAttributes whom="separate instance"/>
</aura:component>
```

Output


Observe! Components within components!
Hello, HTML!
Check out the style in this list.

- I'm red.
- I'm blue.
- I'm green.

Hello passed attribute! Hello separate instance!

Component Body

The root-level tag of every component is `<aura:component>`. Every component inherits the `body` attribute from `<aura:component>`.

The `<aura:component>` tag can contain tags, such as `<aura:attribute>`, `<aura:registerEvent>`, `<aura:handler>`, `<aura:set>`, and so on. Any free markup that is not enclosed in one of the tags allowed in a component is assumed to be part of the body and is set in the `body` attribute.

The `body` attribute has type `Aura.Component[]`. It can be an array of one component, or an empty array, but it's always an array.

In a component, use "`v`" to access the collection of attributes. For example, `{!v.body}` outputs the body of the component.

Setting the Body Content

To set the `body` attribute in a component, add free markup within the `<aura:component>` tag. For example:

```
<aura:component>
    <!--START BODY-->
    <div>Body part</div>
    <lightning:button label="Push Me" onclick="{!c.doSomething}"/>
    <!--END BODY-->
</aura:component>
```

To set the value of an inherited attribute, use the `<aura:set>` tag. Setting the body content is equivalent to wrapping that free markup inside `<aura:set attribute="body">`. Since the `body` attribute has this special behavior, you can omit `<aura:set attribute="body">`.

The previous sample is a shortcut for this markup. We recommend the less verbose syntax in the previous sample.

```
<aura:component>
    <aura:set attribute="body">
        <!--START BODY-->
        <div>Body part</div>
        <lightning:button label="Push Me" onclick="{!c.doSomething}"/>
        <!--END BODY-->
    </aura:set>
</aura:component>
```

The same logic applies when you use any component that has a `body` attribute, not just `<aura:component>`. For example:

```
<lightning:tabset>
    <lightning:tab label="Tab 1">
        Hello world!
    </lightning:tab>
</lightning:tabset>
```

This is a shortcut for:

```
<lightning:tabset>
    <lightning:tab label="Tab 1">
        <aura:set attribute="body">
            Hello World!
        </aura:set>
    </lightning:tab>
</lightning:tabset>
```

Accessing the Component Body

To access a component body in JavaScript, use `component.get("v.body")`.

SEE ALSO:

aura:set

Working with a Component Body in JavaScript

Component Facets

A facet is any attribute of type `Aura.Component[]`. Use this type as a placeholder for a block of markup. The `body` attribute is an example of a facet.

To define your own facet, add an `aura:attribute` tag of type `Aura.Component[]`, which is an array of components, to your component. For example, let's create a component called `facetHeader.cmp`.

```
<!--c:facetHeader-->
<aura:component>
    <aura:attribute name="header" type="Aura.Component[]"/>

    <div>
        <span class="headerClass">{!v.header}</span><br/>
        <span class="bodyClass">{!v.body}</span>
    </div>
</aura:component>
```

This component has a `header` facet. Note how we position the output of the header using the `v.header` expression.

The component doesn't have any output when you access it directly as the `header` and `body` attributes aren't set. Let's create another component, `helloFacets.cmp`, that sets these attributes.

```
<!--c:helloFacets-->
<aura:component>
    See how we set the header facet.<br/>

    <c:facetHeader>

        This is the component body for facetHeader.

        <aura:set attribute="header">
            Hello Header!
        </aura:set>
    </c:facetHeader>

</aura:component>
```

The `aura:set` tag sets the value of the `header` attribute of `facetHeader.cmp`.

The `body` attribute is special. You don't need to use `aura:set` if you're setting the `body` attribute. Any free markup that's not enclosed in one of the tags allowed in a component is assumed to be part of the body and is set in the `body` attribute.

If you use `c:helloFacets` in an app, the output is:

```
See how we set the header facet.
Hello Header!
This is the component body for facetHeader.
```

SEE ALSO:

Component Body

Framework-Specific Types

Controlling Access

The framework enables you to control access to your applications, attributes, components, events, interfaces, and methods via the `access` system attribute. The `access` system attribute indicates whether the resource can be used outside of its own namespace.

Use the `access` system attribute on these tags:

- `<aura:application>`
- `<aura:attribute>`
- `<aura:component>`
- `<aura:event>`
- `<aura:interface>`
- `<aura:method>`

Access Values

You can specify these values for the `access` system attribute.

`private`

Available within the component, app, interface, or event, or method and can't be referenced outside the resource. This value can only be used for `<aura:attribute>`.

Marking an attribute as private makes it easier to refactor the attribute in the future as the attribute can only be used within the resource.

Accessing a private attribute returns `undefined` unless you reference it from the component in which it's declared. You can't access a private attribute from a sub-component that extends the component containing the private attribute.

`public`

Available within your org only. This is the default access value.

`global`

Available in all orgs.

Note: Mark your resources, such as a component, with `access="global"` to make the resource usable outside of your own org. For example, if you want a component to be usable in an installed package or by a Lightning App Builder user or a Experience Builder user in another org.

You can also create documentation for a component, event, or interface marked `access="global"`. This documentation is automatically displayed in the Component Library of an org that uses or installs your package.

Example

This sample component has global access.

```
<aura:component access="global">
    ...
</aura:component>
```

Access Violations

If your code accesses a resource, such as a component, that doesn't have an `access` system attribute allowing you to access the resource:

- Client-side code doesn't execute or returns `undefined`. If you enabled debug mode, you see an error message in your browser console.
- Server-side code results in the component failing to load. If you enabled debug mode, you see a popup error message.

Anatomy of an Access Check Error Message

Here is a sample access check error message for an access violation.

```
Access  Check  Failed ! ComponentService.getDef():'markup://c:targetComponent' is not
visible to 'markup://c:sourceComponent'.
```

An error message has four parts:

1. The context (who is trying to access the resource). In our example, this is `markup://c:sourceComponent`.
2. The target (the resource being accessed). In our example, this is `markup://c:targetComponent`.
3. The type of failure. In our example, this is `not visible`.
4. The code that triggered the failure. This is usually a class method. In our example, this is `ComponentService.getDef()`, which means that the target definition (component) was not accessible. A definition describes metadata for a resource, such as a component.

Fixing Access Check Errors

Tip: If your code isn't working as you expect, enable debug mode to get better error reporting.

You can fix access check errors using one or more of these techniques.

- Add appropriate `access` system attributes to the resources that you own.

- Remove references in your code to resources that aren't available. In the earlier example, `markup://c:targetComponent` doesn't have an access value allowing `markup://c:sourceComponent` to access it.
- Ensure that an attribute that you're accessing exists by looking at its `<aura:attribute>` definition. Confirm that you're using the correct case-sensitive spelling for the `name`.

  Accessing an undefined attribute or an attribute that is out of scope, for example a private attribute, triggers the same access violation message. The access context doesn't know whether the attribute is undefined or inaccessible.

Example: **`is not visible to 'undefined'`**

```
ComponentService.getDef():'markup://c:targetComponent' is not visible to 'undefined'
```

The key word in this error message is `undefined`, which indicates that the framework has lost context. This happens when your code accesses a component outside the normal framework lifecycle, such as in a `setTimeout()` or `setInterval()` call or in an ES6 Promise.

Fix this error by wrapping the code in a `$A.getCallback()` call. For more information, see Modifying Components Outside the Framework Lifecycle.

Example: **`Cannot read property 'Yb' of undefined`**

```
Action failed: c$sourceComponent$controller$doInit [Cannot read property 'Yb' of undefined]
```

This error message happens when you reference a property on a variable with a value of `undefined`. The error can happen in many contexts, one of which is the side-effect of an access check failure. For example, let's see what happens when you try to access an undefined attribute, `imaginaryAttribute`, in JavaScript.

```
var whatDoYouExpect = cmp.get("v.imaginaryAttribute");
```

This is an access check error and `whatDoYouExpect` is set to `undefined`. Now, if you try to access a property on `whatDoYouExpect`, you get an error.

```
Action failed: c$sourceComponent$controller$doInit [Cannot read property 'Yb' of undefined]
```

The `c$sourceComponent$controller$doInit` portion of the error message tells you that the error is in the `doInit` method of the controller of the `sourceComponent` component in the `c` namespace.

IN THIS SECTION:

Application Access Control
The `access` attribute on the `aura:application` tag controls whether the app can be used outside of the app's namespace.

Interface Access Control
The `access` attribute on the `aura:interface` tag controls whether the interface can be used outside of the interface's namespace.

Component Access Control
The `access` attribute on the `aura:component` tag controls whether the component can be used outside of the component's namespace.

Attribute Access Control
The `access` attribute on the `aura:attribute` tag controls whether the attribute can be used outside of the attribute's namespace.

Event Access Control
The `access` attribute on the `aura:event` tag controls whether the event can be used outside of the event's namespace.

SEE ALSO:
Salesforce Help: **Enable Debug Mode for Lightning Components**
Writing Documentation for the Component Library

Application Access Control

The `access` attribute on the `aura:application` tag controls whether the app can be used outside of the app's namespace.

Possible values are listed below.

Modifier	Description
`public`	Available within your org only. This is the default access value.
`global`	Available in all orgs.

Interface Access Control

The `access` attribute on the `aura:interface` tag controls whether the interface can be used outside of the interface's namespace.

Possible values are listed below.

Modifier	Description
`public`	Available within your org only. This is the default access value.
`global`	Available in all orgs.

A component can implement an interface using the `implements` attribute on the `aura:component` tag.

Component Access Control

The `access` attribute on the `aura:component` tag controls whether the component can be used outside of the component's namespace.

Possible values are listed below.

Modifier	Description
`public`	Available within your org only. This is the default access value.
`global`	Available in all orgs.

Note: Components aren't directly addressable via a URL. To check your component output, embed your component in a `.app` resource.

Attribute Access Control

The `access` attribute on the `aura:attribute` tag controls whether the attribute can be used outside of the attribute's namespace.

Possible values are listed below.

Access	Description
`private`	Available within the component, app, interface, or event, or method and can't be referenced outside the resource. **Note:** Accessing a private attribute returns `undefined` unless you reference it from the component in which it's declared. You can't access a private attribute from a sub-component that extends the component containing the private attribute.
`public`	Available within your org only. This is the default access value.
`global`	Available in all orgs.

Event Access Control

The `access` attribute on the `aura:event` tag controls whether the event can be used outside of the event's namespace.

Possible values are listed below.

Modifier	Description
`public`	Available within your org only. This is the default access value.
`global`	Available in all orgs.

Using Object-Oriented Development

The framework provides the basic constructs of inheritance and encapsulation from object-oriented programming and applies them to presentation layer development.

For example, components are encapsulated and their internals stay private. Consumers of the component can access the public shape (attributes and registered events) of the component, but can't access other implementation details in the component bundle. This separation gives component authors freedom to change the internal implementation details and insulates component consumers from those changes.

Inheritance in Aura components doesn't work the way it does in Apex or Java. Using inheritance in Aura makes your code harder to understand as the behavior isn't always intuitive. When possible, use composition instead of inheritance.

IN THIS SECTION:

Abstract Components

Interfaces

Interfaces define a component's shape by defining attributes, events, or methods that any implementing component contains. To use an interface, a component must implement it. An interface can't be used directly in markup.

Inheritance Rules

Favor Composition Over Inheritance

Aura supports inheritance, but it favors composition. When possible, use composition.

When you create an Aura component, you use a mix of inheritance and composition. For example, when you create an application or a component, you extend one of the default base components: `aura:application` or `aura:component`. That's inheritance and it works well. However, when you create a custom component that extends another component, inheritance can get a little more complicated.

Component composition happens when you nest a component in another component. To build a component body, you add components within the body. This component composition enables you to build complex components from simpler building-block components.

Why use composition? Because code reuse and testing become easier.

Also, one feature of inheritance works differently in Aura than it does in most languages and frameworks. This difference makes inheritance lose some of its charm.

When you instantiate a Java class you create one instance, no matter how long that class's inheritance path is. Not so in Aura. Aura creates one instance of the subclassed component and one instance of its parent. The more levels of inheritance, the more component instances are created. Inheritance consumes more memory and processor resources than you might expect.

What is Inherited?

This topic lists what is inherited when you extend a definition, such as a component.

When a component contains another component, we refer in the documentation to parent and child components in the containment hierarchy. When a component extends another component, we refer to sub and super components in the inheritance hierarchy.

Component Attributes

A sub component that extends a super component inherits the attributes of the super component. Use `<aura:set>` in the markup of a sub component to set the value of an attribute inherited from a super component.

Events

A sub component that extends a super component can handle events fired by the super component. The sub component automatically inherits the event handlers from the super component.

The super and sub component can handle the same event in different ways by adding an `<aura:handler>` tag to the sub component. The framework doesn't guarantee the order of event handling.

Helpers

A sub component's helper inherits the methods from the helper of its super component. A sub component can override a super component's helper method by defining a method with the same name as an inherited method.

Controllers

A sub component that extends a super component can call actions in the super component's client-side controller. For example, if the super component has an action called `doSomething`, the sub component can directly call the action using the `{!c.doSomething}` syntax.

Note: We don't recommend using inheritance of client-side controllers as this feature may be deprecated in the future to preserve better component encapsulation. We recommend that you put common code in a helper instead.

SEE ALSO:

- Favor Composition Over Inheritance
- Component Attributes
- Communicating with Events
- Sharing JavaScript Code in a Component Bundle
- Handling Events with Client-Side Controllers
- aura:set

Inherited Component Attributes

A sub component that extends a super component inherits the attributes of the super component.

Attribute values are identical at any level of extension. There is an exception to this rule for the `body` attribute, which we'll look at more closely soon.

Let's start with a simple example. `c:super` has a `description` attribute with a value of "Default description",

```
<!--c:super-->
<aura:component extensible="true">
    <aura:attribute name="description" type="String" default="Default description" />

    <p>super.cmp description: {!v.description}</p>

    {!v.body}
</aura:component>
```

Don't worry about the `{!v.body}` expression for now. We'll explain that when we talk about the `body` attribute.

`c:sub` extends `c:super` by setting `extends="c:super"` in its `<aura:component>` tag.

```
<!--c:sub-->
<aura:component extends="c:super">
    <p>sub.cmp description: {!v.description}</p>
</aura:component
```

Note that `sub.cmp` has access to the inherited `description` attribute and it has the same value in `sub.cmp` and `super.cmp`.

Use `<aura:set>` in the markup of a sub component to set the value of an inherited attribute.

Inherited `body` Attribute

Every component inherits the `body` attribute from `<aura:component>`. The inheritance behavior of `body` is different than other attributes. It can have different values at each level of component extension to enable different output from each component in the inheritance chain. This will be clearer when we look at an example.

Any free markup that is not enclosed in another tag is assumed to be part of the `body`. It's equivalent to wrapping that free markup inside `<aura:set attribute="body">`.

The default renderer for a component iterates through its `body` attribute, renders everything, and passes the rendered data to its super component. The super component can output the data passed to it by including `{!v.body}` in its markup. If there is no super component, you've hit the root component and the data is inserted into `document.body`.

Let's look at a simple example to understand how the `body` attribute behaves at different levels of component extension. We have three components.

`c:superBody` is the super component. It inherently extends `<aura:component>`.

```
<!--c:superBody-->
<aura:component extensible="true">
    Parent body: {!v.body}
</aura:component>
```

At this point, `c:superBody` doesn't output anything for `{!v.body}` as it's just a placeholder for data that will be passed in by a component that extends `c:superBody`.

`c:subBody` extends `c:superBody` by setting `extends="c:superBody"` in its `<aura:component>` tag.

```
<!--c:subBody-->
<aura:component extends="c:superBody">
    Child body: {!v.body}
</aura:component>
```

`c:subBody` outputs:

```
Parent body: Child body:
```

In other words, `c:subBody` sets the value for `{!v.body}` in its super component, `c:superBody`.

`c:containerBody` contains a reference to `c:subBody`.

```
<!--c:containerBody-->
<aura:component>
    <c:subBody>
        Body value
    </c:subBody>
</aura:component>
```

In `c:containerBody`, we set the `body` attribute of `c:subBody` to `Body value`. `c:containerBody` outputs:

```
Parent body: Child body: Body value
```

SEE ALSO:

aura:set

Component Body

Component Markup

Abstract Components

Object-oriented languages, such as Java, support the concept of an abstract class that provides a partial implementation for an object but leaves the remaining implementation to concrete sub-classes. An abstract class in Java can't be instantiated directly, but a non-abstract subclass can.

Similarly, the Aura Components programming model supports the concept of abstract components that have a partial implementation but leave the remaining implementation to concrete sub-components.

To use an abstract component, you must extend it and fill out the remaining implementation. An abstract component can't be used directly in markup.

The `<aura:component>` tag has a boolean `abstract` attribute. Set `abstract="true"` to make the component abstract.

SEE ALSO:

Interfaces

Interfaces

Interfaces define a component's shape by defining attributes, events, or methods that any implementing component contains. To use an interface, a component must implement it. An interface can't be used directly in markup.

An interface starts with the `<aura:interface>` tag, and can contain only these tags:

`<aura:attribute>`

This tag defines an attribute. An interface can have zero or more attributes.

Note: To set the value of an attribute inherited from an interface, redefine the attribute in the sub component using `<aura:attribute>` and set the value in its default attribute. When you extend a component, you can use `<aura:set>` in a sub component to set the value of any attribute that's inherited from the super component. However, this usage of `<aura:set>` doesn't work for attributes inherited from an interface.

`<aura:registerEvent>`

This tag registers an event that can be fired by a component that implements the interface. There's no logic in the interface for firing the event. A component that implements the interface contains the code to fire the event.

`<aura:method>`

This tag defines a method as part of the API of a component that implements the interface. There's no logic for the method in the interface. A component that implements the interface contains the method logic.

You can't use markup, renderers, controllers, or anything else in an interface.

Implement an Interface

To implement an interface, set the `implements` system attribute in the `<aura:component>` tag to the name of the interface that you are implementing. For example:

```
<aura:component implements="mynamespace:myinterface" >
```

A component can implement an interface and extend another component.

```
<aura:component extends="ns1:cmp1" implements="ns2:intf1" >
```

An interface can extend multiple interfaces using a comma-separated list.

```
<aura:interface extends="ns:intf1,ns:int2" >
```

Since there are fewer restrictions on the content of abstract components, they are more common than interfaces. A component can implement multiple interfaces but can only extend one abstract component, so interfaces can be more useful for some design patterns.

Example

Here's an example of an interface.

```
<aura:interface>
    <aura:attribute name="value" type="String"/>

    <aura:registerEvent name="onItemSelected" type="ui:response"
      description="The event fired when the user selects an item" />

    <aura:method name="methodFromInterface">
        <aura:attribute name="stringAttribute" type="String" default="default string"/>
    </aura:method>
</aura:interface>
```

IN THIS SECTION:

Marker Interfaces
A marker interface is an empty interface with no attributes, events, or methods. A marker interface is used to enable specific usage for a component in an app.

SEE ALSO:

Setting Attributes Inherited from an Interface

Abstract Components

Marker Interfaces

A marker interface is an empty interface with no attributes, events, or methods. A marker interface is used to enable specific usage for a component in an app.

For example, a component that implements the `force:appHostable` interface can be used as a custom tab in Lightning Experience or the Salesforce mobile app.

In JavaScript, you can determine if a component implements an interface by using `myCmp.isInstanceOf("mynamespace:myinterface")`.

SEE ALSO:

Configure Components for Custom Tabs

Inheritance Rules

This table describes the inheritance rules for various elements.

Element	extends	implements	Default Base Element
component	one extensible component	multiple interfaces	`<aura:component>`

Element	extends	implements	Default Base Element
app	one extensible app	N/A	`<aura:application>`
interface	multiple interfaces using a comma-separated list (`extends="ns:intf1,ns:int2"`)	N/A	N/A

SEE ALSO:

Interfaces

Best Practices for Conditional Markup

Using the `<aura:if>` tag is the preferred approach to conditionally display markup but there are alternatives. Consider the performance cost and code maintainability when you design components. The best design choice depends on your use case.

Conditionally Create Elements with `<aura:if>`

Let's look at a simple example that shows an error message when an error occurs.

```
<aura:if isTrue="{!v.isError}">
    <div>{!v.errorMessage}</div>
</aura:if>
```

The `<div>` component and its contents are only created and rendered if the value of the `isTrue` expression evaluates to `true`. If the value of the `isTrue` expression changes and evaluates to `false`, all the components inside the `<aura:if>` tag are destroyed. The components are created again if the `isTrue` expression changes again and evaluates to `true`.

The general guideline is to use `<aura:if>` because it helps your components load faster initially by deferring the creation and rendering of the enclosed element tree until the condition is fulfilled.

Toggle Visibility Using CSS

You can use CSS to toggle visibility of markup by calling `$A.util.toggleClass(cmp, 'class')` in JavaScript code.

Elements in markup are created and rendered up front, but they're hidden. For an example, see Dynamically Showing or Hiding Markup.

The conditional markup is created and rendered even if it's not used, so `<aura:if>` is preferred.

Dynamically Create Components in JavaScript

You can dynamically create components in JavaScript code. However, writing code is usually harder to maintain and debug than using markup. Again, using `<aura:if>` is preferred but the best design choice depends on your use case.

SEE ALSO:

Conditional Expressions

Dynamically Creating Components

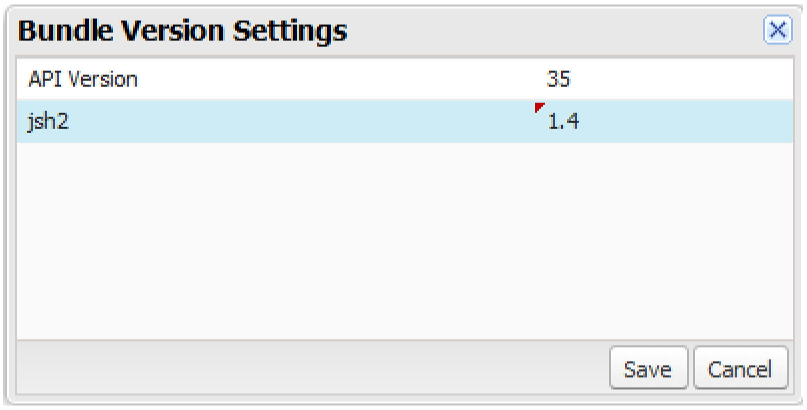
Aura Component Versioning for Managed Packages

Aura component versioning enables you to declare dependencies against specific revisions of an installed managed package.

By assigning a version to your component, you have granular control over how the component functions when new versions of a managed package are released. For example, imagine that a ***`<packageNamespace>`*****`:`**`button` is pinned to version 2.0 of a package. Upon installing version 3.0, the button retains its version 2.0 functionality.

Note: The package developer is responsible for inserting versioning logic into the markup when updating a component. If the component wasn't changed in the update or if the markup doesn't account for version, the component behaves in the context of the most recent version.

Versions are assigned declaratively in the Developer Console. When you're working on a component, click **Bundle Version Settings** in the right panel to define the version. You can only version a component if you've installed a package, and the valid versions for the component are the available versions of that package. Versions are in the format *`<major>`*`.`*`<minor>`*. So if you assign a component version 1.4, its behavior depends on the first major release and fourth minor release of the associated package.



API Version	35
jsh2	1.4

When working with components, you can version:

- Apex controllers
- JavaScript controllers
- JavaScript helpers
- JavaScript renderers
- Bundle markup
  - Applications (`.app`)
  - Components (`.cmp`)
  - Interfaces (`.intf`)
  - Events (`.evt`)

You can't version any other types of resources in bundles. Unsupported types include:

- Styles (`.css`)
- Documentation (`.doc`)
- Design (`.design`)
- SVG (`.svg`)

Once you've assigned versions to components, or if you're developing components for a package, you can retrieve the version in several contexts.

Resource	Return Type	Expression
Apex	Version	`System.requestVersion()`
JavaScript	String	`cmp.getVersion()`
Aura component markup	String	`{!Version}`

You can use the retrieved version to add logic to your code or markup to assign different functionality to different versions. Here's an example of using versioning in an `<aura:if>` statement.

```
<aura:component>
 <aura:if isTrue="{!Version > 1.0}">
  <c:newVersionFunctionality/>
 </aura:if>
 <c:oldVersionFunctionality/>
 ...
</aura:component>
```

SEE ALSO:

Base Components with Minimum API Version Requirements

Security for Lightning Components: Disable Lightning Locker for an Aura Component

Base Components with Minimum API Version Requirements

Some Lightning base components require the custom components that use them to be set to a minimum API version. A custom component's API version must be equal to or later than the latest API version required by any of the components it uses.

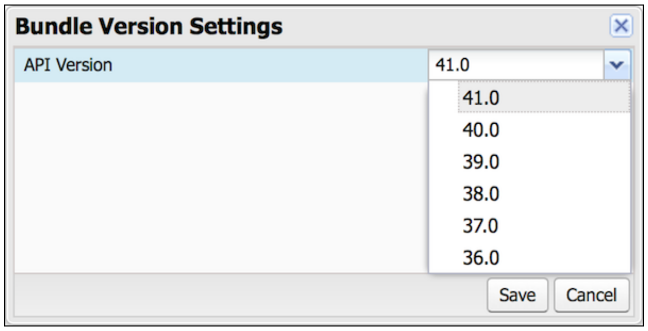
A custom component can become subject to another component's minimum version requirement in several ways.

- The custom component can extend from the component with the minimum version requirement.
- The custom component can add another component as a child component in markup.
- The custom component can dynamically create and add a child component in JavaScript.

If the relationship between components can be determined by static analysis, the version dependency is checked when the component is saved. If a custom component has an API version earlier than a minimum version required by any of the components it uses, an error is reported, and the component isn't saved. Depending on the tool you're using, this error is presented in different ways.

If a component is created dynamically, the relationship between it and its parent component can't be determined at save time. The minimum version requirement is checked at run time, and if it fails a run-time error is reported to the current user.

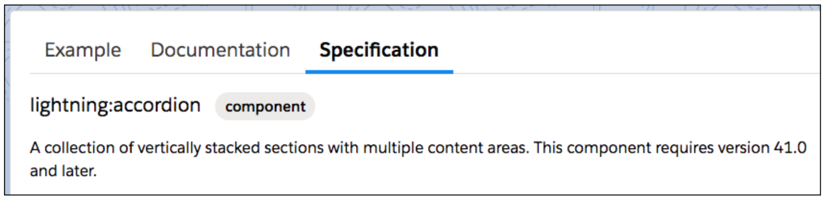
Set the API version for your component in the Developer Console, the Salesforce Extensions for Visual Studio Code, or via API.



Minimum API Version of Lightning Base Components

The minimum API version required to use a base component is listed on the component's Specification page in the Component Library on page 463. Components that don't specify a minimum API version are usable with any API version supported for Lightning components.

For example, `lightning:accordion` requires version 41.0 and later.



A collection of vertically stacked sections with multiple content areas. This component requires version 41.0 and later.

The minimum version for base components that are Generally Available (GA) won't increase in future releases. (However, as with Visualforce components, their behavior might change depending on the API version of the containing component.)

Note: The base components are not versioned. Changing your custom component's API version on the **Bundle Version Settings** window to an earlier version does not impact the behavior of a base component you're using. So if you're using `lightning:map` in a component set to API version 45.0, its behavior does not change if you set your component to 44.0 or 46.0. The latest behavior is observed across all versions.

Deprecation of Lightning Base Components

When a component is deprecated, it's no longer officially supported or tested. However, it's still available for use with any version of the API. The component's behavior is undetermined and could change at any time. The same applies for the deprecation of components, events and interfaces listed in the Component Library on page 463. For example, if a component is deprecated in API version 43.0 (Summer '18), we no longer accept support cases after that release unless otherwise specified.

We recommend you use another component to replace the deprecated component, as described in the reference docs in the Component Library. For example, the deprecated components in the `ui` namespace have been superseded by components in the `lightning` namespace. For more information, see Migrate Components from the ui Namespace on page 122.

Deprecated components may be removed in a future release and should not be relied on. Salesforce does not currently intend to remove deprecated components. However, if that position changes, customers will be given ample warning.

SEE ALSO:

Aura Component Versioning for Managed Packages

Security for Lightning Components: Disable Lightning Locker for an Aura Component

Validations for Aura Component Code

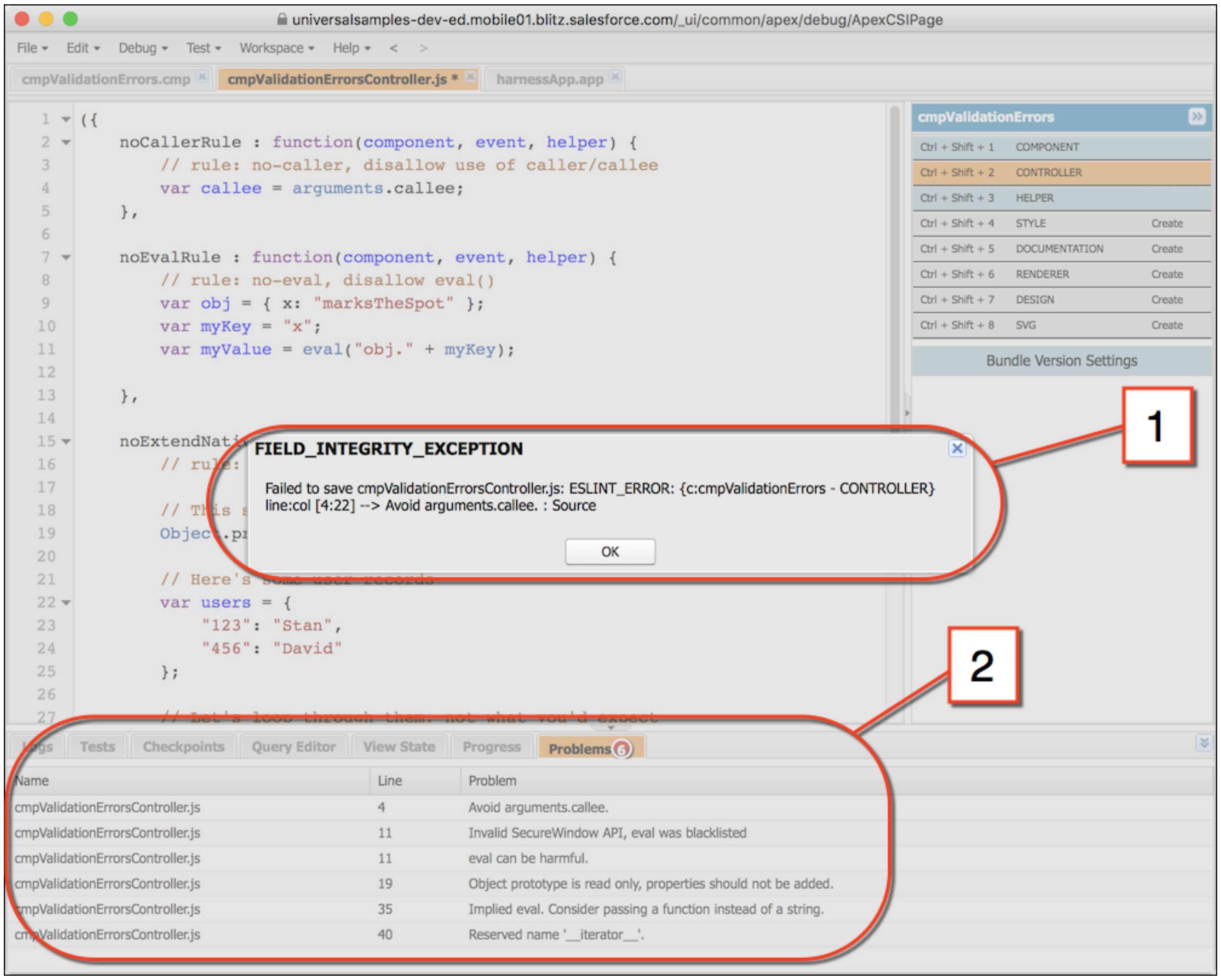
Validate your Aura component code to ensure compatibility with Aura component APIs, best practices, and avoidance of anti-patterns. There are several ways to validate your code. Minimal save-time validations catch the most significant issues only, while Salesforce DX tools provide more comprehensive static code analysis.

IN THIS SECTION:

Validation When You Save Code Changes

Aura component JavaScript code is validated when you save it. Validation ensures that your components are written using best practices and avoid common pitfalls that can make them incompatible with Lightning Locker. Validation happens automatically when you save Aura component resources in the Developer Console, in your favorite IDE, and via API.

Validation failures are treated as errors and block changes from being saved. Error messages explain the failures. Depending on the tool you're using, these errors are presented in different ways. For example, the Developer Console shows an alert for the first error it encounters (1), and lists all of the validation errors discovered in the **Problems** tab (2).


universalsamples-dev-ed.mobile01.blitz.salesforce.com/_ui/common/apex/debug/ApexCSIPage
File Edit Debug Test Workspace Help < >
cmpValidationErrors.cmp
cmpValidationErrorsController.js *
harnessApp.app
1
2
Logs Tests Checkpoints Query Editor View State Progress Problems 6
OK

```
1  ({
2      noCallerRule : function(component, event, helper) {
3          // rule: no-caller, disallow use of caller/callee
4          var callee = arguments.callee;
5      },
6
7      noEvalRule : function(component, event, helper) {
8          // rule: no-eval, disallow eval()
9          var obj = { x: "marksTheSpot" };
10         var myKey = "x";
11         var myValue = eval("obj." + myKey);
12
13     },
14
15     noExtendNati
16         // rule:
17
18         // This s
19         Object.pr
20
21         // Here's some user records
22         var users = {
23             "123": "Stan",
24             "456": "David"
25         };
26
27         // Let's loop through them: not what you'd expect
```

cmpValidationErrors		
Ctrl + Shift + 1	COMPONENT	
Ctrl + Shift + 2	CONTROLLER	
Ctrl + Shift + 3	HELPER	
Ctrl + Shift + 4	STYLE	Create
Ctrl + Shift + 5	DOCUMENTATION	Create
Ctrl + Shift + 6	RENDERER	Create
Ctrl + Shift + 7	DESIGN	Create
Ctrl + Shift + 8	SVG	Create

Bundle Version Settings

FIELD_INTEGRITY_EXCEPTION

Failed to save cmpValidationErrorsController.js: ESLINT_ERROR: {c:cmpValidationErrors - CONTROLLER} line:col [4:22] --> Avoid arguments.callee. : Source

Name	Line	Problem
cmpValidationErrorsController.js	4	Avoid arguments.callee.
cmpValidationErrorsController.js	11	Invalid SecureWindow API, eval was blacklisted
cmpValidationErrorsController.js	11	eval can be harmful.
cmpValidationErrorsController.js	19	Object prototype is read only, properties should not be added.
cmpValidationErrorsController.js	35	Implied eval. Consider passing a function instead of a string.
cmpValidationErrorsController.js	40	Reserved name '__iterator__'.

Validations are applied only to components set to API version 41.0 and later. If the validation service prevents you from saving important changes, set the component version to API 40.0 or earlier to disable validations temporarily. When you've corrected the coding errors, return your component to API 41.0 or later to save it with passing validations.

Validation During Development Using ESLint

Use ESLint to scan and improve your code during development. A linting tool doesn't just help you avoid Lightning Locker conflicts and anti-patterns. It's a terrific practice for improving your code quality and consistency, and to uncover subtle bugs before you commit them to your codebase.

Validations using ESLint are done separately from saving your code to Salesforce. The results are informational only.

Salesforce DX used to include a `force:lightning:lint` command but the command was removed in February, 2022. Instead, we recommend the Aura plugin for ESLint for linting.

To install the Aura plugin for ESLint, see this GitHub repo.

Validations performed using the Aura plugin for ESLint are different from validations performed at save time in the following important ways.

- ESLint uses many more rules to analyze your component code. Save-time validations prevent you from making the most fundamental mistakes only. Validation with ESLint errs on the side of giving you more information.
- Validation via ESLint ignores the API version of your components. Save-time validations are performed only for components set to API 41.0 and later.

Aura Component Validation Rules

Rules built into Aura component code validations cover restrictions under Lightning Locker, correct use of Lightning APIs, and a number of best practices for writing Aura component code. Each rule, when triggered by your code, points to an area where your code might have an issue.

In addition to the Lightning-specific rules we've created, other rules are active in Lightning validations, included from ESLint basic rules. Documentation for these rules is available on the ESLint project site. If you encounter an error or warning from a rule not described here, search for it on the ESLint Rules page.

The set of rules used to validate your code varies depending on the tool you use, and the way you use it. Minimal save-time validations catch the most significant issues only, while Salesforce DX tools provide more comprehensive static code analysis.

IN THIS SECTION:

Disallow Use of __iterator__ Property (no-iterator)

Prevents using the obsolete `__iterator__` property. Use standard JavaScript iterators and generators instead. This is a standard rule built into ESLint.

Disallow Use of __proto__ (no-proto)

Prevents using the obsolete `__proto__` property, which was deprecated in ECMAScript 3.1. Use `Object.getPrototypeOf()` instead. This is a standard rule built into ESLint.

Disallow with Statements (no-with)

Prevents using `with` statements, which adds members of an object to the current scope in a way that makes it hard to predict or view impact or behavior. This is a standard rule built into ESLint.

Validation Rules Used at Save Time

The following rules are used for validations that are done when you save your Aura component code.

Validation failures for any of these rules prevents saving changes to your code.

Lightning Platform Rules

These rules are specific to Aura component JavaScript code. These custom rules are written and maintained by Salesforce.

Validate Aura API (aura-api)

This rule verifies that use of the framework APIs is according to the published documentation. The use of undocumented or private features is disallowed.

Validate Secure Document Public API (secure-document)

This rule validates that only supported functions and properties of the `document` global are accessed.

Validate Secure Window Public API (secure-window)

This rule validates that only supported functions and properties of the `window` global are accessed.

Validate JavaScript Intrinsic APIs (`ecma-intrinsics`)

This rule deals with the intrinsic APIs in JavaScript, more formally known as ECMAScript.

When Lightning Locker is enabled, the framework prevents the use of unsupported API objects or calls. That means your Aura components code is allowed to use:

- Features built into JavaScript ("intrinsic" features)
- Published, supported features built into the Aura Components programming model.
- Published, supported features built into Lightning Locker *SecureObject* objects

What exactly are these "intrinsic APIs"? They're the APIs defined in the ECMAScript Language Specification. That is, things built into JavaScript. This includes Annex B of the specification, which deals with legacy browser features that aren't part of the "core" of JavaScript, but are nevertheless still supported for JavaScript running inside a web browser.

Note that some features of JavaScript that you might consider intrinsic—for example, the `window` and `document` global variables—are superceded by *SecureObject* objects, which offer a more constrained API.

Rule Details

This rule verifies that use of the intrinsic JavaScript APIs is according to the published specification. The use of non-standard, deprecated, and removed language features is disallowed.

Further Reading

- ECMAScript specification
- Annex B: Additional ECMAScript Features for Web Browsers
- Intrinsic Objects (JavaScript)

SEE ALSO:

Validate Aura API (aura-api)

Validate Aura Component Public API (secure-component)

Validate Secure Document Public API (secure-document)

Validate Secure Window Public API (secure-window)

Validate Aura API (`aura-api`)

This rule verifies that use of the framework APIs is according to the published documentation. The use of undocumented or private features is disallowed.

When Lightning Locker is enabled, the framework prevents the use of unsupported API objects or calls. That means your Aura components code is allowed to use:

- Features built into JavaScript ("intrinsic" features)
- Published, supported features built into the Aura Components programming model.
- Published, supported features built into Lightning Locker *SecureObject* objects

This rule deals with the supported, public framework APIs, for example, those available through the framework global `$A`.

Why is this rule called "Aura API"? Because the core of the Aura Components programming model is the open source Aura Framework. And this rule verifies permitted uses of that framework, rather than anything specific to Lightning Components.

Rule Details

The following patterns are considered problematic:

```
Aura.something(); // Use $A instead
$A.util.fake(); // fake is not available in $A.util
```

Further Reading

For details of all of the methods available in the framework, including `$A`, see the JavaScript API documentation on page 472.

SEE ALSO:

Validate Aura Component Public API (secure-component)

Validate Secure Document Public API (secure-document)

Validate Secure Window Public API (secure-window)

Validate Aura Component Public API (`secure-component`)

This rule validates that only public, supported framework API functions and properties are used.

When Lightning Locker is enabled, the framework prevents the use of unsupported API objects or calls. That means your Aura components code is allowed to use:

- Features built into JavaScript ("intrinsic" features)
- Published, supported features built into the Aura Components programming model.
- Published, supported features built into Lightning Locker *SecureObject* objects

Prior to Lightning Locker, when you created or obtained a reference to a component, you could call any function and access any property available on that component, even if it wasn't public. When Lightning Locker is enabled, components are "wrapped" by a new SecureComponent object, which controls access to the component and its functions and properties. SecureComponent restricts you to using only the published, supported Component API.

SEE ALSO:

Validate Aura API (aura-api)

Validate Secure Document Public API (secure-document)

Validate Secure Window Public API (secure-window)

Validate Secure Document Public API (`secure-document`)

This rule validates that only supported functions and properties of the `document` global are accessed.

When Lightning Locker is enabled, the framework prevents the use of unsupported API objects or calls. That means your Aura components code is allowed to use:

- Features built into JavaScript ("intrinsic" features)
- Published, supported features built into the Aura Components programming model.
- Published, supported features built into Lightning Locker *SecureObject* objects

Prior to Lightning Locker, when you accessed the `document` global, you could call any function and access any property available. When Lightning Locker is enabled, the `document` global is "wrapped" by a new SecureDocument object, which controls access to `document` and its functions and properties. SecureDocument restricts you to using only "safe" features of the `document` global.

SEE ALSO:

Validate Aura API (aura-api)

Validate Aura Component Public API (secure-component)

Validate Secure Window Public API (secure-window)

Validate Secure Window Public API (`secure-window`)

This rule validates that only supported functions and properties of the `window` global are accessed.

When Lightning Locker is enabled, the framework prevents the use of unsupported API objects or calls. That means your Aura components code is allowed to use:

- Features built into JavaScript ("intrinsic" features)
- Published, supported features built into the Aura Components programming model.
- Published, supported features built into Lightning Locker *SecureObject* objects

Prior to Lightning Locker, when you accessed the `window` global, you could call any function and access any property available. When Lightning Locker is enabled, the `window` global is “wrapped” by a new SecureWindow object, which controls access to `window` and its functions and properties. SecureWindow restricts you to using only “safe” features of the `window` global.

SEE ALSO:

Validate Aura API (aura-api)

Validate Aura Component Public API (secure-component)

Validate Secure Document Public API (secure-document)

Disallow Use of `caller` and `callee` (`no-caller`)

Prevent the use of `arguments.caller` and `arguments.callee`. These are also forbidden in ECMAScript 5 and later when in strict mode, which is enabled under Lightning Locker. This is a standard rule built into ESLint.

For complete details about this rule, including examples, see the corresponding ESLint documentation, Disallow Use of caller/callee (no-caller).

Disallow Script URLs (`no-script-url`)

Prevents the use of `javascript:` URLs. This is a standard rule built into ESLint.

For complete details about this rule, including examples, see the corresponding ESLint documentation, Disallow Script URLs (no-script-url).

Disallow Extending Native Objects (`no-extend-native`)

Prevent changing the behavior of built-in JavaScript objects, such as Object or Array, by modifying their prototypes. This is a standard rule built into ESLint.

For complete details about this rule, including examples, see the corresponding ESLint documentation, Disallow Extending of Native Objects (no-extend-native).

Disallow Calling Global Object Properties as Functions (`no-obj-calls`)

Prevents calling the `Math`, `JSON`, and `Reflect` global objects as though they were functions. For example, `Math()` is disallowed. This follows the ECMAScript 5 specification. This is a standard rule built into ESLint.

For complete details about this rule, including examples, see the corresponding ESLint documentation, disallow calling global object properties as functions (no-obj-calls).

Disallow Use of `__iterator__` Property (`no-iterator`)

Prevents using the obsolete `__iterator__` property. Use standard JavaScript iterators and generators instead. This is a standard rule built into ESLint.

For complete details about this rule, including examples, see the corresponding ESLint documentation, Disallow Iterator (no-iterator).

Disallow Use of `__proto__` (`no-proto`)

Prevents using the obsolete `__proto__` property, which was deprecated in ECMAScript 3.1. Use `Object.getPrototypeOf()` instead. This is a standard rule built into ESLint.

For complete details about this rule, including examples, see the corresponding ESLint documentation, Disallow Use of __proto__ (no-proto).

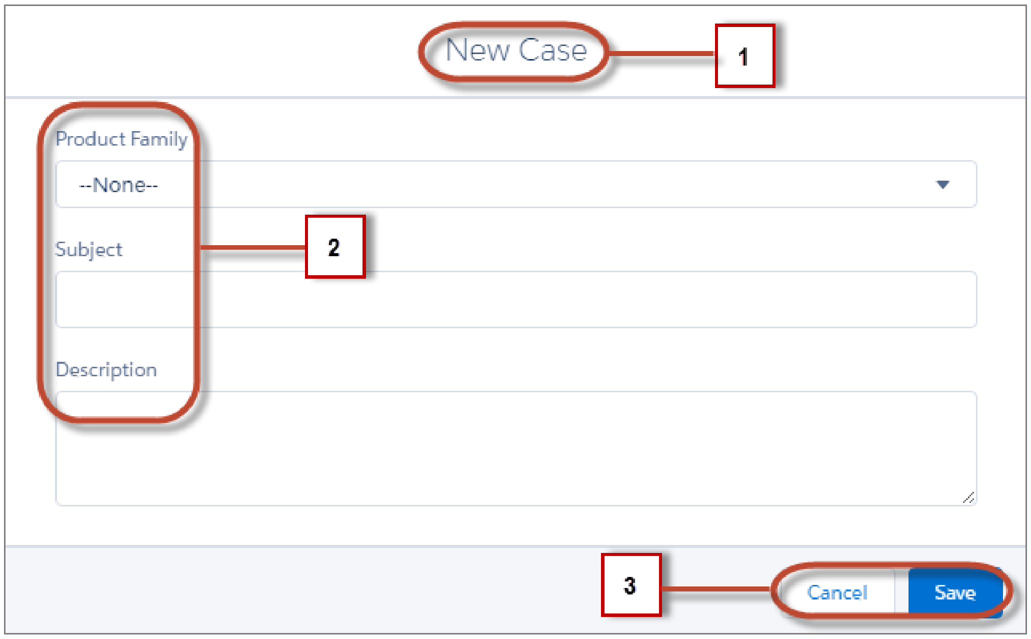
Disallow `with` Statements (`no-with`)

Prevents using `with` statements, which adds members of an object to the current scope in a way that makes it hard to predict or view impact or behavior. This is a standard rule built into ESLint.

For complete details about this rule, including examples, see the corresponding ESLint documentation, disallow with statements (no-with).

Using Labels

Labels are text that presents information about the user interface, such as in the header (1), input fields (2), or buttons (3). While you can specify labels by providing text values in component markup, you can also access labels stored outside your code using the `$Label` global value provider in expression syntax.



This section discusses how to use the `$Label` global value provider in these contexts:

- The `label` attribute in input components
- The `format()` expression function for dynamically populating placeholder values in labels

IN THIS SECTION:

Using Custom Labels
Custom labels are custom text values that can be translated into any language that Salesforce supports. To access custom labels in Aura components, use the `$Label` global value provider.

Input Component Labels
A label describes the purpose of an input component. To set a label on an input component, use the `label` attribute.

Dynamically Populating Label Parameters
Output and update labels using the `format()` expression function.

Getting Labels in JavaScript

You can retrieve labels in JavaScript code. Your code performs optimally if the labels are statically defined and sent to the client when the component is loaded.

Getting Labels in Apex

You can retrieve label values in Apex code and set them on your component using JavaScript.

Setting Label Values via a Parent Attribute

Setting label values via a parent attribute is useful if you want control over labels in child components.

Using Custom Labels

Custom labels are custom text values that can be translated into any language that Salesforce supports. To access custom labels in Aura components, use the `$Label` global value provider.

Custom labels enable developers to create multilingual applications by automatically presenting information (for example, help text or error messages) in a user's native language.

Note: Label translations require the Translation Workbench is enabled.

To create custom labels, from Setup, enter *`Custom Labels`* in the `Quick Find` box, then select **Custom Labels**.

Use the following syntax to access custom labels in Aura components.

- `$Label.c.`***`labelName`*** for the default namespace
- `$Label.`***`namespace`***`.`***`labelName`*** if your org has a namespace, or to access a label in a managed package

You can reference custom labels in component markup and in JavaScript code. Here are some examples.

Label in a markup expression using the default namespace

`{!$Label.c.`***`labelName`***`}`

Note: Label expressions in markup are supported in `.cmp` and `.app` resources only.

Label in JavaScript code if your org has a namespace

`$A.get("$Label.`***`namespace`***`.`***`labelName`***`")`

Note: Updates to a label locale or translation are not immediately in the application. To verify the change immediately, log out and in.

SEE ALSO:

Value Providers

Salesforce Help: Translate Custom Labels

Input Component Labels

A label describes the purpose of an input component. To set a label on an input component, use the `label` attribute.

This example shows how to use labels using the `label` attribute on an input component.

```
<lightning:input type="number" name="myNumber" label="Pick a Number:" value="54" />
```

The label is placed on the left of the input field and can be hidden by setting `variant="label-hidden"`, which applies the `slds-assistive-text` class to the label to support accessibility.

Using `$Label`

Use the `$Label` global value provider to access labels stored in an external source. For example:

```
<lightning:input type="number" name="myNumber" label="{!$Label.Number.PickOne}" />
```

To output a label and dynamically update it, use the `format()` expression function. For example, if you have `np.labelName` set to `Hello {0}`, the following expression returns `Hello World` if `v.name` is set to `World`.

```
{!format($Label.np.labelName, v.name)}
```

SEE ALSO:

Supporting Accessibility

Dynamically Populating Label Parameters

Output and update labels using the `format()` expression function.

You can provide a string with placeholders, which are replaced by the substitution values at runtime.

Add as many parameters as you need. The parameters are numbered and are zero-based. For example, if you have three parameters, they will be named `{0}`, `{1}`, and `{2}`, and they will be substituted in the order they're specified.

Let's look at a custom label, `$Label.mySection.myLabel`, with a value of `Hello {0} and {1}`, where `$Label` is the global value provider that accesses your labels.

This expression dynamically populates the placeholder parameters with the values of the supplied attributes.

```
{!format($Label.mySection.myLabel, v.attribute1, v.attribute2)}
```

The label is automatically refreshed if one of the attribute values changes.

Note: Always use the `$Label` global value provider to reference a label with placeholder parameters. You can't set a string with placeholder parameters as the first argument for `format()`. For example, this syntax doesn't work:

```
{!format('Hello {0}', v.name)}
```

Use this expression instead.

```
{!format($Label.mySection.salutation, v.name)}
```

where `$Label.mySection.salutation` is set to `Hello {0}`.

Getting Labels in JavaScript

You can retrieve labels in JavaScript code. Your code performs optimally if the labels are statically defined and sent to the client when the component is loaded.

Static Labels

Static labels are defined in one string, such as `"$Label.c.task_mode_today"`. The framework parses static labels in markup or JavaScript code and sends the labels to the client when the component is loaded. A server trip isn't required to resolve the label.

Use `$A.get()` to retrieve static labels in JavaScript code. For example:

```
var staticLabel = $A.get("$Label.c.task_mode_today");
component.set("v.mylabel", staticLabel);
```

You can also retrieve label values using Apex code and send them to the component via JavaScript code. For more information, see Getting Labels in Apex.

Dynamic Labels

`$A.get(`***`labelReference`***`)` must be able to resolve the label reference at compile time, so that the label values can be sent to the client along with the component definition.

If you must defer label resolution until runtime, you can dynamically create labels in JavaScript code. This technique can be useful when you need to use a label, but which specific label isn't known until runtime.

```
// Assume the day variable is dynamically generated
// earlier in the code
// THIS CODE WON'T WORK
var dynamicLabel = $A.get("$Label.c." + day);
```

If the label is already known on the client, `$A.get()` displays the label. If the value is not known, an empty string is displayed in production mode, or a placeholder value showing the label key is displayed in debug mode.

Using `$A.get()` with a label that can't be determined at runtime means that `dynamicLabel` is an empty string, and won't be updated to the retrieved value. Since the label, `"$Label.c." + day`, is dynamically generated, the framework can't parse it or send it to the client when the component is requested.

There are a few alternative approaches to using `$A.get()` so that you can work with dynamically generated labels.

If your component uses a known set of dynamically constructed labels, you can avoid a server roundtrip for the labels by adding a reference to the labels in a JavaScript resource. The framework sends these labels to the client when the component is requested. For example, if your component dynamically generates `$Label.c.task_mode_today` and `$Label.c.task_mode_tomorrow` label keys, you can add references to the labels in a comment in a JavaScript resource, such as a client-side controller or helper.

```
// hints to ensure labels are preloaded
// $Label.c.task_mode_today
// $Label.c.task_mode_tomorrow
```

If your code dynamically generates many labels, this approach doesn't scale well.

If you don't want to add comment hints for all the potential labels, the alternative is to use `$A.getReference()`. This approach comes with the added cost of a server trip to retrieve the label value.

This example dynamically constructs the label value by calling `$A.getReference()` and updates a `tempLabelAttr` component attribute with the retrieved label.

```
var labelSubStr = "task_mode_today";
var labelReference = $A.getReference("$Label.c." + labelSubStr);
cmp.set("v.tempLabelAttr", labelReference);
var dynamicLabel = cmp.get("v.tempLabelAttr");
```

`$A.getReference()` returns a reference to the label. This **isn't** a string, and you shouldn't treat it like one. You never get a string label directly back from `$A.getReference()`.

Instead, use the returned reference to set a component's attribute value. Our code does this in `cmp.set("v.tempLabelAttr", labelReference);`.

When the label value is asynchronously returned from the server, the attribute value is automatically updated as it's a reference. The component is rerendered and the label value displays.

Note: Our code sets `dynamicLabel = cmp.get("v.tempLabelAttr")` immediately after getting the reference. This code displays an empty string until the label value is returned from the server. If you don't want that behavior, use a comment hint to ensure that the label is sent to the client without requiring a later server trip.

SEE ALSO:

Using JavaScript

Input Component Labels

Dynamically Populating Label Parameters

Getting Labels in Apex

You can retrieve label values in Apex code and set them on your component using JavaScript.

Custom Labels

Custom labels have a limit of 1,000 characters and can be accessed from an Apex class. To define custom labels, from Setup, in the Quick Find box, enter *`Custom Labels`*, and then select **Custom Labels**.

In your Apex class, reference the label with the syntax `System.Label.`***`MyLabelName`***.

```
public with sharing class LabelController {
    @AuraEnabled
    public static String getLabel() {
        String s1 = 'Hello from Apex Controller, ' ;
        String s2 = System.Label.MyLabelName;
        return s1 + s2;
    }
}
```

Note: Return label values as plain text strings. You can't return a label expression using the `$Label` global value provider.

The component loads the labels by requesting it from the server, such as during initialization. For example, the label is retrieved in JavaScript code.

```
({
    doInit : function(component, event, helper) {
        var action = component.get("c.getLabel");
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                component.set("v.mylabel", response.getReturnValue());
            }
            // error handling when state is "INCOMPLETE" or "ERROR"
         });
         $A.enqueueAction(action);
    }
})
```

Finally, make sure you wire up the Apex class to your component. The label is set on the component during initialization.

```
<aura:component controller="LabelController">
    <aura:handler name="init" value="{!this}" action="{!c.doInit}" />
    <aura:attribute name="mylabel" type="String"/>
    {!v.mylabel}
</aura:component>
```

Passing in Label Values

Pass label values into components using the expression syntax `{!v.mylabel}`. You must provide a default value to the String attribute. Depending on your use case, the default value might be the label in the default language or, if the specific label can't be known until runtime, perhaps just a single space.

```
<aura:component controller="LabelController">
    <aura:attribute name="mylabel" type="String" default=" "/>
    <lightning:input name="mytext" label="{!v.mylabel}"/>
</aura:component>
```

You can also retrieve labels in JavaScript code, including dynamically creating labels that are generated during runtime. For more information, see Getting Labels in JavaScript.

Retrieving Custom Labels Using `System.Label` Methods

You can use methods in the `System.Label` class to check for and retrieve translated labels.

- To check if translation exists for a label and language in a namespace, use `translationExists(namespace, label, language)`.
- To retrieve the custom label for a specified namespace and default language setting, use `get(namespace, label)`.
- To retrieve the custom label for a specified namespace and language, use `get(namespace, label, language)`.

SEE ALSO:

Apex Reference Guide: Label Class

Setting Label Values via a Parent Attribute

Setting label values via a parent attribute is useful if you want control over labels in child components.

Let's say that you have a container component, which contains another component, `inner.cmp`. You want to set a label value in `inner.cmp` via an attribute on the container component. This can be done by specifying the attribute type and default value. You must set a default value in the parent attribute if you are setting a label on an inner component, as shown in the following example.

This is the container component, which contains a default value `My Label` for the `_label` attribute .

```
<aura:component>
    <aura:attribute name="_label"
                    type="String"
                    default="My Label"/>
    <lightning:button label="Set Label" aura:id="button1" onclick="{!c.setLabel}"/>
    <auradocs:inner aura:id="inner" label="{!v._label}"/>
</aura:component>
```

This inner component contains a text area component and a label attribute that's set by the container component.

```
<aura:component>
    <aura:attribute name="label" type="String"/>
    <lightning:textarea aura:id="textarea"
                      name="myTextarea"
                      label="{!v.label}"/>
</aura:component>
```

This client-side controller action updates the label value.

```
({
    setLabel:function(cmp) {
        cmp.set("v._label", 'new label');
    }
})
```

When the component is initialized, you'll see a button and a text area with the label My Label. When the button in the container component is clicked, the setLabel action updates the label value in the inner component. This action finds the label attribute and sets its value to new label.

SEE ALSO:

Input Component Labels

Component Attributes

Localization

The framework provides client-side localization support on input and output components.

You can use the global value provider, $Locale, to obtain the locale information. The locale setting in your organization overrides the browser's locale information. Base Lightning components adapt automatically to the language, locale, and time zone settings of the Salesforce org they run in.

Working with Locale, Language, and Timezone

In a single currency organization, Salesforce administrators set the currency locale, default language, default locale, and default time zone for their organizations. Users can set their individual language, locale, and time zone on their personal settings pages.

Note: Single language organizations cannot change their language, although they can change their locale.

If you're working with Salesforce data, we recommend using the base components built on Lightning Data Service. For example, the lightning:recordForm and lightning:recordViewForm components can display a read-only value of your record data. See Lightning Data Service on page 384.

Consider the lightning:formatted* components only if the lightning:record*Form components don't meet your requirements.

Let's take a look at the lightning:formattedDateTime component to display a date and time. Setting the time zone on the Language & Time Zone page to (GMT+02:00) returns the date and time like Sep 28, 2020, 11:13 AM when you run the following code.

```
<lightning:formattedDateTime value="2020-09-28T18:13:41Z"
            year="numeric" month="short" day="2-digit"
            hour="2-digit"
            minute="2-digit"/>
```

Changing the user locale to `French (France)` returns the date and time like `28 sept. 2020 à 11:13`. Running `$A.get("$Locale.userLocaleCountry")` returns the user's locale, for example, `FR`.

For more information, see Supported Locales and ICU Formats.

To display a currency value, use `lightning:formattedNumber`. Setting the currency locale on the Company Information page to `Japanese (Japan) - JPY` returns `¥100,000` when you run the following code.

```
<lightning:formattedNumber value="100000" style="currency" />
```

Similarly, running `$A.get("$Locale.currency")` returns `"¥"` when your org's currency locale is set to `Japanese (Japan) - JPY`. For more information, see Supported Locales and ICU Formats" in the Salesforce Help.

Working with Address, Name, and Number Formats

The user's locale determines the name and address formats. Numbers, including currency, decimal, and percentage are also formatted according to the user's locale. See Supported Locales and ICU Formats.

To get user input for an address, use `lightning:inputAddress`. For a read-only output of an address, use `lightning:formattedAddress`. The default output displays an address that links to Google Maps.

To get user input for a person's name, use `lightning:inputName`. For a read-only output of a name, use `lightning:formattedName`.

To get user input for a number, including currency, decimal, and percentage, use `lightning:input` with `number` type. For a read-only output of a number, use `lightning:formattedNumber`.

Working with Date and Time Formats

To get user input for a date and time, use `lightning:input` with type `date`, `datetime`, or `time`. To customize the date and time formats, we recommend using `lightning:formattedDateTime` or `lightning:formattedTime`.

This example sets the date and time from a Date object using the `init` handler. The `timeZone` attribute is optional and is used to override the default timezone based on the user's location.

```
<aura:component>
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
    <aura:attribute name="datetime" type="DateTime"/>
    <lightning:formattedDateTime value="{!v.datetime}" timeZone="Europe/Berlin"
        year="numeric" month="short" day="2-digit" hour="2-digit"
        minute="2-digit" second="2-digit"/>
 </aura:component>
```

```
({
    doInit : function(component, event, helper) {
        var date = new Date();
        component.set("v.datetime", date);
    }
})
```

This example renders the date in the format `MMM DD, YYYY HH:MM:SS AM`. Refer to the component reference for examples on how you can display the date in a different format.

SEE ALSO:

Formatting Dates in JavaScript

Working with Base Lightning Components

Base Lightning components are the building blocks that make up the modern user interfaces in Lightning Experience, Salesforce app, and Experience Builder sites.

Base Lightning components incorporate Lightning Design System markup and classes, providing improved performance and accessibility with a minimum footprint.

These base components handle the details of HTML and CSS for you. Each component provides simple attributes that enable variations in style. This means that you typically don't need to use CSS at all. The simplicity of the base Lightning component attributes and their clean and consistent definitions make them easy to use, enabling you to focus on your business logic.

You can find base Lightning components in the `lightning` namespace to complement the existing `ui` namespace components. In instances where there are matching `ui` and `lightning` namespace components, we recommend that you use the `lightning` namespace component. The `lightning` namespace components are optimized for common use cases. Beyond being equipped with the Lightning Design System styling, they handle accessibility, real-time interaction, and enhanced error messages.

Note: Components in the `lightning` namespace are available in two versions—as Aura components and Lightning web components. We recommend using Lightning web components whenever possible. Lightning web components are custom HTML elements built using HTML and modern JavaScript. Lightning web components and Aura components can coexist and interoperate on a page. To admins and end users, they both appear as Lightning components. See the *Lightning Web Components Developer Guide* for more information.

In subsequent releases, we intend to provide additional base Lightning components. We expect that in time the `lightning` namespace will have parity with the `ui` namespace and go beyond it. In addition, the base Lightning components will evolve with the Lightning Design System over time. This ensures that your customizations continue to match Lightning Experience and the Salesforce mobile app.

While the base components are visual building blocks and provides minimum functionality out-of-the-box, they can be combined together to build "experience components" with richer capabilities and made accessible via the Lightning App Builder. Admins can drag-and-drop these experience components to build and configure user interfaces easily. For example, the Chatter Feed component in Lightning App Builder comprises a collection of tabs, a group of buttons, and a rich text editor.

The API version column denotes the minimum API version you must set to use the component in the Developer Console, the Salesforce Extensions for Visual Studio Code, or via API. Components that don't specify a minimum API version are usable with any API version 37.0 and later.

Note: Interactive examples for the following components are available in the Component Library.

Buttons

These components provide different button flavors.

Type	Lightning Component Name	Description	Lightning Design System	API Version
Button	`lightning:button`	Represents a button element.	Buttons	

Type	Lightning Component Name	Description	Lightning Design System	API Version
Button Group	`lightning:buttonGroup`	Represents a group of buttons.	Button Groups	
Button Icon	`lightning:buttonIcon`	An icon-only HTML button.	Button Icons	
Button Icon (Stateful)	`lightning:buttonIconStateful`	An icon-only button that retains state.	Button Icons	41.0
Button Menu	`lightning:buttonMenu`	A dropdown menu with a list of actions or functions.	Menus	
	`lightning:menuItem`	A list item in `lightning:buttonMenu`.		
	`lightning:menuDivider`	A horizontal line separating menu items in `lightning:buttonMenu`.		
	`lightning:menuSubheader`	A subheading for menu items in `lightning:buttonMenu`.		
Button Stateful	`lightning:buttonStateful`	A button that toggles between states.	Button Stateful	
Insert Image Button	`lightning:insertImageButton`	A button for inserting images in `lightning:inputRichText`.	Button Icons	43.0

Data Entry

Use these components for data entry.

Type	Lightning Component Name	Description	Lightning Design System	API Version
Address	`lightning:inputAddress`	Represents an address compound field.	Form Layout	42.0
Checkbox Group	`lightning:checkboxGroup`	Enables single or multiple selection on a group of options.	Checkbox	41.0
Combobox	`lightning:combobox`	An input element that enables single selection from a list of options.	Combobox	
Dual Listbox	`lightning:dualListbox`	Provides an input listbox accompanied with a listbox of selectable options. Options can be moved between the two lists.	Dueling Picklist	41.0
File Uploader and Preview	`lightning:fileUpload`	Enables file uploads to a record.	File Selector	41.0
	`lightning:fileCard`	Displays a representation of uploaded content.	Files	

Type	Lightning Component Name	Description	Lightning Design System	API Version
Input	`lightning:input`	Represents interactive controls that accept user input depending on the type attribute.	Input	
Input Field	`lightning:inputField`	Represents an editable input for a field on a Salesforce object.	Form Layout	42.0
Input Name	`lightning:inputName`	Represents a name compound field.	Form Layout	42.0
Input Location (Geolocation)	`lightning:inputLocation`	A geolocation compound field that accepts a latitude and longitude value.	Form Layout	41.0
Radio Group	`lightning:radioGroup`	Enables single selection on a group of options.	Radio Button Radio Button Group	41.0
Select	`lightning:select`	Creates an HTML `select` element.	Select	
Slider	`lightning:slider`	An input range slider for specifying a value between two specified numbers.	Slider	41.0
Rich Text Area	`lightning:inputRichText`	A WYSIWYG editor with a customizable toolbar for entering rich text.	Rich Text Editor	
Text Area	`lightning:textArea`	A multiline text input.	Textarea	

Displaying Data

Use these components to display data.

Type	Lightning Component Name	Description	Lightning Design System	API Version
Address	`lightning:formattedAddress`	Displays a formatted address that provides a link to the given location on Google Maps.	N/A	42.0
Click-to-dial	`lightning:clickToDial`			
Date/Time	`lightning:formattedDateTime`	Displays formatted date and time.		
Email	`lightning:formattedEmail`	Displays an email as a hyperlink with the `mailto:` URL scheme.		41.0
Geolocation	`lightning:formattedLocation`	Displays a geolocation using the format `latitude, longitude`.		41.0

Type	Lightning Component Name	Description	Lightning Design System	API Version
Name	`lightning:formattedName`	Displays a formatted name that can include a salutation and suffix.		42.0
Number	`lightning:formattedNumber`	Displays formatted numbers.		
Output Field	`lightning:outputField`	Displays a label, help text, and value for a field on a Salesforce object.		41.0
Phone	`lightning:formattedPhone`	Displays a phone number as a hyperlink with the `tel:` URL scheme.		41.0
Rich Text	`lightning:formattedRichText`	Displays rich text that's formatted with allowed tags and attributes.		41.0
Text	`lightning:formattedText`	Displays text, replaces newlines with line breaks, and linkifies if requested.		41.0
Time	`lightning:formattedTime`	Displays a formatted time based on the user's locale.		42.0
URL	`lightning:formattedUrl`	Displays a URL as a hyperlink.		41.0
Relative Date/Time	`lightning:relativeDateTime`	Displays the relative time difference between the source date-time and the provided date-time.		

Forms

use these components to edit and view records

Type	Lightning Component Name	Description	Lightning Design System	API Version
Record Edit Form	`lightning:recordEditForm`	A grouping of `lightning:inputField` components of record fields to be edited in a form.	Form Layout	42.0
Record Form	`lightning:recordForm`	A container to simplify form creation for viewing and editing record fields.	Form Layout	43.0
Record View Form	`lightning:recordViewForm`	A grouping of `lightning:outputField` components and other formatted display components to display record fields in a form.	Form Layout	42.0

Layout

The following components group related information together.

Type	Lightning Component Name	Description	Lightning Design System	API Version
Accordion	`lightning:accordion`	A collection of vertically stacked sections with multiple content areas.	Accordion	41.0
	`lightning:accordionSection`	A single section that is nested in a `lightning:accordion` component.		
Card	`lightning:card`	Applies a container around a related grouping of information.	Cards	
Carousel	`lightning:carousel`	A collection of images that are displayed horizontally one at a time.	Carousel	42.0
Layout	`lightning:layout`	Responsive grid system for arranging containers on a page.	Grid	
	`lightning:layoutItem`	A container within a `lightning:layout` component.		
Tabs	`lightning:tab`	A single tab that is nested in a `lightning:tabset` component.	Tabs	
	`lightning:tabset`	Represents a list of tabs.		
Tile	`lightning:tile`	A grouping of related information associated with a record.	Tiles	

Navigation Components

The following components organize links and actions in a hierarchy or to visit other locations in an app.

Type	Lightning Component Name	Description	Lightning Design System	API Version
Breadcrumb	`lightning:breadcrumb`	An item in the hierarchy path of the page the user is on.	Breadcrumbs	
	`lightning:breadcrumbs`	A hierarchy path of the page you're currently visiting within the website or app.		
Navigation	`lightning:navigation`	Generates a URL for a page reference.	N/A	43.0
Map	`lightning:map`	Displays a map of one or more locations	Map	44.0
Tree	`lightning:tree`	Displays a structural hierarchy with nested items.	Trees	41.0
Vertical Navigation	`lightning:verticalNavigation`	A vertical list of links that take you to another page or parts of the page you're in.	Vertical Navigation	41.0
	`lightning:verticalNavigationItem`	A text-only link within `lightning:verticalNavigationSection` or `lightning:verticalNavigationOverflow`		

Type	Lightning Component Name	Description	Lightning Design System	API Version
	`lightning:verticalNavigationItemBadge`	A link and badge within `lightning:verticalNavigationSection` or `lightning:verticalNavigationOverflow`		
	`lightning:verticalNavigationItemIcon`	A link and icon within `lightning:verticalNavigationSection` or `lightning:verticalNavigationOverflow`		
	`lightning:verticalNavigationOverflow`	An overflow of items in `lightning:verticalNavigation`		
	`lightning:verticalNavigationSection`	A section within `lightning:verticalNavigation`		

Visual Components

The following components provide informative cues, for example, like icons and loading spinners.

Type	Lightning Component Name	Description	Lightning Design System	API Version
Avatar	`lightning:avatar`	A visual representation of an object.	Avatars	
Badge	`lightning:badge`	A label that holds a small amount of information.	Badges	
Data Table	`lightning:datatable`	A table that displays columns of data, formatted according to type.	Data Tables	41.0
Dynamic Icon	`lightning:dynamicIcon`	A variety of animated icons.	Dynamic Icons	41.0
Help Text (Tooltip)	`lightning:helptext`	An icon with a popover container a small amount of text.	Tooltips	
Icon	`lightning:icon`	A visual element that provides context.	Icons	
List View	`lightning:listView`	Displays a list view of the specified object	N/A	42.0
Messaging	`lightning:overlayLibrary`	Displays messages via modals and popovers.	Messaging	41.0
	`lightning:notificationsLibrary`	Displays messages via notices and toasts.	Messaging	41.0
Path	`lightning:path`	Displays a path driven by a picklist field and Path Setup metadata.	Path	41.0
Picklist Path	`lightning:picklistPath`	Displays a path based on a specified picklist field.		

Type	Lightning Component Name	Description	Lightning Design System	API Version
Pill	`lightning:pill`	A pill represents an existing item in a database, as opposed to user-generated freeform text.	Pills	
Pill Container	`lightning:pillContainer`	A list of pills grouped in a container.	Pills	42.0
Progress Bar	`lightning:progressBar`	A horizontal progress bar from left to right to indicate the progress of an operation.	Progress Bars	41.0
Progress Indicator and Path	`lightning:progressIndicator`	Displays steps in a process to indicate what has been completed.	Progress Indicators Path	41.0
Tree Grid	`lightning:treeGrid`	A hierarchical view of data presented in a table.	Trees	42.0
Spinner	`lightning:spinner`	Displays an animated spinner.	Spinners	

Feature-Specific Components

The following components are usable only in the context of specific Salesforce features.

Type	Lightning Component Name	Description	Lightning Design System	API Version
Lightning Page Region	`lightning:flexipageRegionInfo`	Provides Lightning page region information to the component that contains it.	N/A	41.0
Flow Interview	`lightning:flow`	Represents a flow interview in Lightning runtime.		41.0
Omni-Channel Toolkit	`lightning:omniToolkitAPI`	Provides API access to methods for the Omni-channel toolkit.		41.0
Lightning Console Utility Bar	`lightning:utilityBarAPI`	Provides API access to methods for the utility bar in Lightning Console.		41.0
Lightning Console Workspace	`lightning:workspaceAPI`	Provides API access to methods for the workspace in Lightning Console.		41.0
Embedded Chat	`lightningsnapin:prechatAPI`	Enables customization of the user interface for the pre-chat page.		42.0
Embedded Chat	`lightningsnapin:settingsAPI`	Provides API access to Embedded Service settings within your custom Embedded Service components.		43.0

Type	Lightning Component Name	Description	Lightning Design System	API Version
Embedded Chat	lightningsnapin:minimizedAPI	Enables customization of the user interface for the minimized chat window for Embedded Chat.		43.0
EmpJs Streaming API library	`lightning:empApi`	Provides access to methods for subscribing to a streaming channel and listening to event messages		44.0

Base Lightning Components Considerations

Learn about the guidelines on using the base Lightning components.

Warning: Don't depend on the markup of a Lightning component as its internals can change in future releases. Reaching into the component internals can also cause unrecoverable errors in the app. For example, using `cmp.get("v.body")` and examining the DOM elements can cause issues in your code if the component markup changes down the road. See the knowledge article Salesforce Component Internals Are Protected.

Alternatives to Targeting the Component DOM

With Lightning Locker enforced, you can't traverse the DOM for components you don't own. Instead of accessing the DOM tree, take advantage of value binding with component attributes and use component methods that are available to you. For example, to get an attribute on a component, use `cmp.find("myInput").get("v.name")` instead of `cmp.find("myInput").getElement().name`. The latter doesn't work if you don't have access to the component, such as a component in another namespace.

Deprecated Block-Element-Modifier (BEM) Notation

In LWC API version 61.0, Salesforce changed the specification for SLDS class name modifiers. We changed from using the double-dash `--` notation for modifiers to using the single-underscore notation `_`. Use the latest notation when authoring your components.

```
<!-- Incorrect BEM notation -->
<a class="slds-button slds-button--neutral" href="#">Neutral Link</a>
```

```
<!-- Correct BEM notation -->
<a class="slds-button slds-button_neutral" href="#">Neutral Link</a>
```

Individual Component Considerations

Many of the base Lightning components are still evolving and these considerations can help you while you're building your apps.

lightning:buttonMenu

This component contains menu items that are created only if the button is triggered. You can't reference the menu items during initialization or if the button isn't triggered yet.

lightning:input

Fields for percentage and currency input must specify a step increment of 0.01 as required by the native implementation.

```
<lightning:input type="number" name="percentVal" label="Enter a percentage value"
formatter="percent" step="0.01" />
<lightning:input type="number" name="currencyVal" label="Enter a dollar amount"
formatter="currency" step="0.01" />
```

When working with checkboxes, radio buttons, and toggle switches, use `aura:id` to group and traverse the array of components. Grouping them enables you to use `get("v.checked")` to determine which elements are checked or unchecked without reaching into the DOM. You can also use the `name` and `value` attributes to identify each component during the iteration. The following example groups three checkboxes together using `aura:id`.

```
<aura:component>
    <form>
      <fieldset>
        <legend>Select your favorite color:</legend>
        <lightning:input type="checkbox" label="Red"
            name="color1" value="1" aura:id="colors"/>
        <lightning:input type="checkbox" label="Blue"
            name="color2" value="2" aura:id="colors"/>
        <lightning:input type="checkbox" label="Green"
            name="color3" value="3" aura:id="colors"/>
      </fieldset>
    <lightning:button label="Submit" onclick="{!c.submitForm}"/>
    </form>
</aura:component>
```

In your client-side controller, you can retrieve the array using `cmp.find("colors")` and inspect the `checked` values.

When working with `type="file"`, you must provide your own server-side logic for uploading files to Salesforce. Read the file using the FileReader HTML object, and then encode the file contents before sending them to your Apex controller. In your Apex controller, you can use the `EncodingUtil` methods to decode the file data. For example, you can use the Attachment object to upload files to a parent object. In this case, you pass in the base64 encoded file to the `Body` field to save the file as an attachment in your Apex controller.

Uploading files using this component is subject to regular Apex controller limits, which is 1 MB. To accommodate file size increase due to base64 encoding, we recommend that you set the maximum file size to 750 KB. You must implement chunking for file sizes larger than 1 MB. Files uploaded via chunking are subject to a size limit of 4 MB. For more information, see the *Apex Developer Guide*. Alternatively, you can use `lightning:fileUpload` to upload files directly to records.

lightning:tab

This component creates its body during runtime. You can't reference the component during initialization. Referencing the component using `aura:id` can return unexpected results, such as the component returning an undefined value when implementing `cmp.find("myComponent")`.

lightning:tabset

When you load more tabs than can fit the width of the viewport, the tabset provides navigation buttons to scroll horizontally to display the overflow tabs.

Methods with Limited Support on Some Components

Some methods have limited support or no support on these components:

- `lightning:avatar`

- `lightning:badge`
- `lightning:breadcrumb`
- `lightning:formattedDateTime`
- `lightning:formattedNumber`
- `lightning:icon`
- `lightning:input`
- `lightning:inputField`
- `lightning:outputField`
- `lightning:relativeDateTime`
- `lightning:textarea`

`getDef()`

`getDef()` can't get API methods or attach change handlers in the specified components. The correct way to work with base Lightning components is to work with the instances, and not attempt to access the component or its definition.

`getReference()`

`getReference()` method support is limited. You can't use it with controllers with these components. For example, `getReference('v.value')` works but `getReference('c.myFunc')` doesn't work.

`afterRender()`

`afterRender()` isn't supported by the specified components. Don't call `afterRender()` on Lightning base components directly. For example, `component.find('lightning:input').afterRender()` doesn't work.

No nested component access

You can't access sub-components inside a base Lightning component. You can only use the exposed API. For example, `cmp.find('mylightning:inputField').find('innercomponent')` doesn't work.

Event Handling in Base Lightning Components

Base components are lightweight and closely resemble HTML markup. They follow standard HTML practices by providing event handlers as attributes, such as `onfocus`, instead of registering and firing Lightning component events, like components in the `ui` namespace.

Because of their markup, you might expect to access DOM elements for base components via `event.target` or `event.currentTarget`. However, this type of access breaks encapsulation because it provides access to another component's DOM elements, which are subject to change.

Use the methods described here to make your code compliant with Lightning Locker.

We recommend binding your value to an attribute. For example, bind the value for `lightning:input` to a `textvalue` attribute.

```
<aura:component>
    <aura:attribute name="textvalue" type="String" default="Initial value"/>
    <lightning:input value="{!v.textvalue}" onchange="{!c.handleInputChange}"/>
</aura:component>
```

In your client-side controller, use the event handler to get the `textvalue` attribute value.

```
({
     handleInputChange : function(component, event) {
        let val = component.get("v.textvalue");
    }
})
```

Alternatively, to retrieve the component that fired the event, use `event.getSource()`.

```
<aura:component>
    <lightning:button name="myButton" label="Click me" onclick="{!c.doSomething}"/>
</aura:component>
```

For an event fired by a custom component or a base Lightning component, use `event.getSource()`. For events fired by standard HTML elements, you can use `event.currentTarget` and `event.target`. For example, `event.target` returns null when the `lightning:button` component in the example is clicked.

```
({
    doSomething: function(cmp, event, helper) {
        var button = event.getSource();

        //Don’t use the following patterns for events
        //that are fired by another component
        //or a base Lightning component
        var el = event.target;
        var currentEl = event.currentTarget;
    }
})
```

Note: If Lightning Web Security is enabled in your org, `event.currentTarget` and `event.target` return the corresponding elements. For example, in the `lightning:button` example, both properties return the `<button>` element.

Retrieve a component attribute that’s passed to the event by using this syntax.

```
event.getSource().get("v.name")
```

Reusing Event Handlers

`event.getSource()` helps you determine which component fired an event. Let’s say you have several buttons that reuse the same `onclick` handler. To retrieve the name of the button that fired the event, use `event.getSource().get("v.name")`.

```
<aura:component>
    <lightning:button label="New Record" name="new" onclick="{!c.handleClick}"/>
    <lightning:button label="Edit" name="edit" onclick="{!c.handleClick}"/>
    <lightning:button label="Delete" name="delete" onclick="{!c.handleClick}"/>
</aura:component>
```

```
({
    handleClick: function(cmp, event, helper) {
        //returns "new", "edit", or "delete"
        var buttonName = event.getSource().get("v.name");
    }
})
```

Retrieving the Active Component Using the `onactive` Handler

When working with tabs, you want to know which one is active. The `lightning:tab` component enables you to obtain a reference to the target component when it becomes active using the `onactive` handler. Clicking the component multiple times invokes the handler one time only.

```
<aura:component>
    <lightning:tabset>
      <lightning:tab onactive="{! c.handleActive }" label="Tab 1" id="tab1" />
      <lightning:tab onactive="{! c.handleActive }" label="Tab 2" id="tab2" />
    </lightning:tabset>
</aura:component>
```

```
({
      handleActive: function (cmp, event) {
          var tab = event.getSource();
          switch (tab.get('v.id')) {
              case 'tab1':
                  //do something when tab1 is clicked
                  break;
              case 'tab2':
                  //do something when tab2 is clicked
                  break;
          }
      }
})
```

Retrieving the ID and Value Using the `onselect` Handler

Some components provide event handlers to pass in events to child components, such as the `onselect` event handler on the following components.

- `lightning:buttonMenu`
- `lightning:tabset`

Although the `event.detail` syntax continues to be supported, we recommend that you update your JavaScript code to use the following patterns for the `onselect` handler as we plan to deprecate `event.detail` in a future release.

- `event.getParam("id")`
- `event.getParam("value")`

For example, you want to retrieve the value of a selected menu item in a `lightning:buttonMenu` component from a client-side controller.

```
//Before
var menuItem = event.detail.menuItem;
var itemValue = menuItem.get("v.value");
//After
var itemValue = event.getParam("value");
```

Similarly, to retrieve the ID of a selected tab in a `lightning:tabset` component:

```
//Before
var tab = event.detail.selectedTab;
var tabId = tab.get("v.id");
```

```
//After
var tabId = event.getParam("id");
```

Note: If you need a reference to the target component, use the `onactive` event handler instead.

Base Components Limitations for Native HTML Events

All supported event handlers on a base component are listed in the Specification tab of the component reference. The event handler names start with `on`, for example, `onchange`.

Base components generally don't support native HTML events, unlike their Lightning web component counterparts. You might encounter unexpected behavior if you try to handle an HTML event on a base component. Let's say you want to handle the `onkeydown` HTML event on `lightning:input`.

```
<aura:component>
    <aura:attribute name="value" type="String" default="Initial value"/>

    <!-- Don't use event handlers that are not supported -->
    <lightning:input value="{!v.value}" onkeydown="{!c.handleKeyDown}"/>
</aura:component>
```

Since `onkeydown` is not a supported event handler based on the `lightning:input` specifications, `event.getParam("value")` and `event.detail` return `undefined`.

Creating a Form

Work with user input for server-side use, such as creating or updating a record. Or get user input to update the user interface, such as displaying or hiding components.

If you're creating a form to work with Salesforce data, use the `lightning:recordForm`, `lightning:recordEditForm`, `lightning:recordViewForm`, or `force:recordData` base components as they are built on Lightning Data Service. Otherwise, you must wire up the fields to the Salesforce object yourself and use Apex to process the user input as shown in the next section.

Example: The Aura Components Basics Trailhead module walks you through building a form for creating an expense record.

Implement a Basic Form

Before proceeding, we recommend that you have working knowledge of web forms, as the rest of the topic builds on that concept.

You can collect data in fields that accept different types of user input, such as a checkbox, date, email, file, password, number, phone, radio, or text. Most user input can be collected by using lightning:input.

Here's a list of form controls for option selection and their corresponding base components.

- Button: `lightning:button` (and `lightning:buttonIcon` and so on)
- Checkbox: `lightning:checkboxGroup`
- Dropdown menu for single selection: `lightning:combobox`
- Dropdown menu for single selection using the HTML `<select>`: `lightning:select`
- Dual listbox for multiple selection: `lightning:dualListbox`
- Radio button: `lightning:radioGroup`

Here's a list of form controls for entering an input value and their corresponding base components.

- Input field: `lightning:input`
- Address compound field: `lightning:inputAddress`
- Geolocation compound field: `lightning:inputLocation`
- Name compound field: `lightning:inputName`
- Rich text field: `lightning:inputRichText`
- Input range for number selection: `lightning:slider`
- Text input (multi-line): `lightning:textarea`

When you use the base components, the `<label>` and `<input>` elements are automatically configured for you. For form styling, you get the Salesforce Lightning Design System (SLDS) styling. You can also use SLDS utility classes to customize the layout of your form.

Let's say we want a form that collects a contact's name, email address, and comments.

In this example, we are using `lightning:inputName`, `lightning:input`, and `lightning:textarea` in a standalone app. To create a grid layout for the fields, use `lightning:layout`.

```
<aura:application access="GLOBAL" extends="force:slds" controller="ContactController">
    <aura:attribute name="salutationOptions" type="List" default="[
        {'label': 'Mr.', 'value': 'Mr.'},
        {'label': 'Ms.', 'value': 'Ms.'},
        {'label': 'Mrs.', 'value': 'Mrs.'},
        {'label': 'Dr.', 'value': 'Dr.'},
        {'label': 'Prof.', 'value': 'Prof.'},
    ]"/>
    <aura:attribute name="newContact" type="Contact"
        default="{ 'sobjectType': 'Contact',
                   'Title': '',
                   'FirstName': '',
                   'LastName': '',
                   'Email': '',
                   'Description': '' }" />
    <aura:attribute name="message" type="String" default=""/>

    <lightning:card iconName="standard:contact" title="Add a Contact">
        <div class="slds-p-around_medium">
            <lightning:layout>
                <lightning:layoutItem size="4" padding="around-small">
                    <lightning:inputName aura:id="contact"
                                         label="Contact Name"
                                         firstName="{!v.newContact.FirstName}"
                                         lastName="{!v.newContact.LastName}"
                                         salutation="{!v.newContact.Title}"
                                         options="{!v.salutationOptions}"
                                         required="true"/>
                </lightning:layoutItem>
                <lightning:layoutItem size="8" padding="around-small">
                    <lightning:input aura:id="contact" label="Email" type="email"
value="{!v.newContact.Email}"/>
                    <lightning:textarea aura:id="contact" label="Comments"
value="{!v.newContact.Description}"/>
                    <lightning:button label="Create Contact"
onclick="{!c.handleCreateContact}" variant="brand" class="slds-m-top_medium"/>
```

```
                </lightning:layoutItem>
            </lightning:layout>
            <p>{!v.message}</p>
        </div>
    </lightning:card>
</aura:application>
```

The client-side controller submits the user data to the Apex controller and updates the `v.message` attribute when the contact is created successfully.

```
({
    handleCreateContact: function(component, event) {
        var saveContactAction = component.get("c.createContact");
            saveContactAction.setParams({
                "contact": component.get("v.newContact")
            });

        // Configure the response handler for the action
        saveContactAction.setCallback(this, function(response) {
            var state = response.getState();
            if(state === "SUCCESS") {
                component.set("v.message", "Contact created successfully");
            }
            else if (state === "ERROR") {
                console.log('Problem saving contact, response state: ' + state);
            }
            else {
                console.log('Unknown problem, response state: ' + state);
            }
        });

        // Send the request to create the new contact
        $A.enqueueAction(saveContactAction);
    },
})
```

The Apex controller uses the `upsert` DML operation to create a contact record.

```
public with sharing class ContactController {

    @AuraEnabled
    public static Contact createContact(Contact contact){
        upsert contact;
        return contact;
    }
}
```

Notice that the form allows you to submit empty fields without any user interaction. The field-level errors for required fields that you leave empty are displayed only after you interact with the fields. Also, if you enter an invalid email format, the email field displays an error.

Customize the submission behavior to prevent invalid fields from getting submitted. For more information, see Validating Fields on page 117.

Validating Fields

Validate user input, handle errors, and display error messages on input fields.

Built-in field validation is available for the base components discussed in Creating a Form on page 114.

Base components simplify input validation by providing attributes to define error conditions, enabling you to handle errors by checking the component's validity state. For example, you can set a minimum length for a field, display an error message when the condition is not met, and handle the error based on the given validity state.

Most of the base components provide attributes and methods to enable different ways to validate input.

Note: Refer to the Component Library for component examples, specification, and documentation.

Require a field

When you set `required="true"`, the field is invalid when a user interacts with it but does not make a selection or enter an input.

Specify a type

A `lightning:input` field that expects a certain data type is invalid if an incorrect data format is entered. For example, the `email` type on `lightning:input` expects an email address and the `number` type expects a number.

Specify a criteria

A `lightning:input` field that specifies a certain criteria or attribute, `max`, `min`, `pattern`, and so on, is invalid if the criteria isn't met. You can provide a custom error message using attributes like `messageWhenValueMissing`, when it's available on the component.

Check field validity

The `validity` attribute on the base components returns the validity states of an input. This attribute is based on the ValidityState object from the Web API. For example, you want to check if a field is valid when a user removes focus from the field.

```
<lightning:input name="input" aura:id="myinput" label="Enter some text" onblur="{!
c.handleBlur }" />
```

If all constraint validations are met, the field returns true .

```
handleBlur: function (cmp, event) {
    var validity = cmp.find("myinput").get("v.validity");
    console.log(validity.valid); //returns true
}
```

Report field validity

To programmatically set and display an error message on a field, use the `setCustomValidity()` and `reportValidity()` methods available on the base components. For more information, see the lightning:input documentation.

Prevent Invalid Fields from Getting Submitted

In Creating a Form on page 114 we implemented a basic form with built-in validation for required fields and specific types. Let's customize the submission behavior such that the form displays errors on invalid fields. Customized behavior is useful if a user tries to submit an empty form and to identify errors in a field.

```
({
    handleCreateContact: function(component, event) {
        var allValid = component.find('contact').reduce(function (validSoFar, inputCmp) {

            inputCmp.reportValidity();
            return validSoFar && inputCmp.checkValidity();
        }, true);

        if (allValid) {
        /******** Insert code from "Creating a Form" topic ********/
        var saveContactAction = component.get("c.createContact");
            saveContactAction.setParams({
                "contact": component.get("v.newContact")
            });

        // Configure the response handler for the action
            saveContactAction.setCallback(this, function(response) {
                var state = response.getState();
                if(state === "SUCCESS") {
                    component.set("v.message", "Contact created successfully");
                }
                else if (state === "ERROR") {
                    console.log('Problem saving contact, response state: ' + state);
                }
                else {
                    console.log('Unknown problem, response state: ' + state);
                }
            });

            // Send the request to create the new contact
            $A.enqueueAction(saveContactAction);
            /******** End code from "Creating a Form" topic ********/
        } else {
            alert('Please update the invalid form entries and try again.');
        }

    },
})
```

Example: The Aura Components Basics Trailhead module walks you through building a form for creating an expense record.

Field Validation Considerations

Client-side field validation provides an initial check for user data before submitting it to the server. Implement your own server-side validation to ensure that user data is saved in the expected format. Consider the following guidelines.

Start with Lightning Data Service

When working with Salesforce data, we recommend that you use the `lightning:recordForm`, `lightning:recordEditForm`, `lightning:recordViewForm`, or `force:recordData` base components. They are built on Lightning Data Service, which ensures data consistency, while handling sharing rules and field-level security for you. The components also provide field validation and error handling.

Enforce data integrity

For data integrity, enforce it at the lowest level possible. For example, specifying `required="true"` on the base components is only cosmetic. Enforce the field as required in the field definition. See Require Field Input to Ensure Data Quality.

Consider validation rules

Define validation rules to verify that the data a user enters in a record meets the standards you specify before the user can save the record. You can retrieve the error that's returned by a validation rule using `response.getError()`. If you don't handle the error, the form submission fails silently.

Alternatively, use a base component built on Lightning Data Service to automatically display the validation rule error on a field. For example, you can define a validation rule that enforces an email address to contain `@example.com`. The field displays the error message if the validation rule fails.

Lightning Design System Considerations

Although the base Lightning components provide Salesforce Lightning Design System styling out-of-the-box, you may still want to write some CSS depending on your requirements.

If you're using the components outside of the Salesforce mobile app and Lightning Experience, such as in standalone apps and Lightning Out, extend `force:slds` to apply Lightning Design System styling to your components. Here are several guidelines for using Lightning Design System in base components.

Using Utility Classes in Base Components

Lightning Design System utility classes are the fundamentals of your component's visual design and promote reusability, such as for alignments, grid, spacing, and typography. Most base components inherits a `class` attribute, so you can add a utility class or custom class to the outer element of the components. For example, you can apply a spacing utility class to `lightning:button`.

```
<lightning:button name="submit" label="Submit" class="slds-m-around_medium"/>
```

The class you add is appended to the base classes that the component includes automatically, resulting in the following rendered element.

```
<button class="slds-button slds-button_neutral slds-m-around_medium"
        type="button" name="submit">Submit</button>
```

Similarly, you can create a custom class by adding it to the CSS resource in the component bundle and pass it into the `class` attribute.

```
<lightning:badge label="My badge" class="myCustomClass"/>
```

You have the flexibility to customize the components at a granular level beyond the CSS scaffolding we provide. Let’s look at the `lightning:card` component, where you can create your own body markup. You can apply the `slds-p-horizontal_small` or `slds-card__body_inner` class in the body markup to add padding around the body.

```
<!-- lightning:card example using slds-p-horizontal_small class -->
<lightning:card>
  <aura:set attribute="title">My Account</aura:set>
  <aura:set attribute="footer">Footer</aura:set>
  <aura:set attribute="actions">
    <lightning:button label="New"/>
  </aura:set>
  <p class="slds-p-horizontal_small">
    Card Body
  </p>
  </lightning:card>
```

```
<!-- lightning:card example using slds-card__body_inner -->
<lightning:card>
    <aura:set attribute="title">My Account</aura:set>
    <aura:set attribute="footer">Footer</aura:set>
    <aura:set attribute="actions">
      <lightning:button label="New"/>
    </aura:set>
      <div class="slds-card__body_inner">
        Card Body
      </div>
</lightning:card>
```

Block-Element-Modifier (BEM) Notation

CSS class names that are used by Lightning base components match the Block-Element-Modifier (BEM) notation that Salesforce Lightning Design System uses. Starting in LWC API version 61.0, class names that previously contained a double dash now use a single underscore in place of the double dash. For example, a CSS class that used to be `slds-p-around--small` is now `slds-p-around_small`. If you have custom CSS in your components that reference an SLDS class that contains a double dash, update your selectors to use a single underscore. See Lightning Design System FAQ.

Applying Custom Component Styling

Sometimes the utility classes aren't enough and you want to add custom styling in your component bundle. You saw earlier that you can create a custom class and pass it into the `class` attribute. We recommend that you create a class instead of targeting a class name you don't own, since those classes might change anytime. For example, don't try to target `.slds-input` or `.lightningInput`, as they are CSS classes that are available by default in base components. You can also consider using tokens to ensure that your design is consistent across your components. Specify values in the token bundle and reuse them in your components' CSS resources.

Showing and Hiding with Visibility Classes

Lightning Design System utility classes include visibility classes that enable you to show and hide elements. These classes are designed as show/hide pairs that you add and remove, or toggle, with JavaScript. Apply only one class at a time. See Lightning Design System: Utilities: Visibility for descriptions of the classes. For information about using JavaScript to toggle markup see Dynamically Showing or Hiding Markup.

Using the Grid for Layout

`lightning:layout` is your answer for a flexible grid system. You can achieve a simple layout by enclosing `lightning:layoutItem` components within `lightning:layout`, which creates a `div` container with the `slds-grid` class. To apply additional Lightning Design System grid classes, specify any combination of the `lightning:layout` attributes. For example, specify `verticalAlign="stretch"` to append the `slds-grid_vertical-stretch` class. You can apply Lightning Design System grid classes to the component using the `horizontalAlign`, `verticalAlign`, and `pullToBoundary` attributes. However, not all grid classes are available through these attributes. To provide additional grid classes, use the `class` attribute. The following grid classes can be added using the `class` attribute.

- `.slds-grid_frame`
- `.slds-grid_vertical`
- `.slds-grid_reverse`
- `.slds-grid_vertical-reverse`
- `.slds-grid_pull-padded-x-small`
- `.slds-grid_pull-padded-xx-small`
- `.slds-grid_pull-padded-xxx-small`

This example adds the `slds-grid_reverse` class to the `slds-grid` class to reverse the horizontal visual flow of the grid columns.

```
<lightning:layout horizontalAlign="space" class="slds-grid_reverse">
   <lightning:layoutItem padding="around-small">
     <!-- more markup here -->
   </lightning:layoutItem>
   <!-- more lightning:layoutItem components here -->
</lightning:layout>
```

For more information, see Lightning Design System: Utilities: Grid.

SEE ALSO:

Styling Apps

Styling with Design Tokens and Styling Hooks

Working with Lightning Design System Variants

Base component variants correspond to blueprint variations in Lightning Design System. Variants change the appearance of a component and are controlled by the `variant` attribute.

Applying Variants to Base Components

Variants on a component refer to design variations for that component, which enable you to change the appearance of the component easily. We try to create variants for each component to apply the design of variations and examples from the SLDS component blueprint. However, not all variants are implemented yet. Most base components provide a `variant` attribute that accepts two or more variants. For example, `lightning:button` supports many variants to apply different text and background colors on the buttons.

This example creates a button with the brand variant.

```
<lightning:button variant="brand" label="Brand" onclick="{! c.handleClick }" />
```

If you don't specify a variant or you pass in a variant that's not supported, the default variant is used instead. For button, the neutral variant is used by default.

Some components don't support a `variant` attribute, but you can use Lightning Design System classes to achieve the styling you want.

This example uses a Lightning Design System class to apply a padding to a paragraph in the `lightning:card` component.

```
<lightning:card footer="Card Footer" title="Hello">
    <aura:set attribute="actions">
        <lightning:button label="New"/>
    </aura:set>
    <p class="slds-p-horizontal_small">
        Card Body with a Lightning Design System class
    </p>
</lightning:card>
```

If you don't find a variant you need, see if you can pass in a Lightning Design System class to the base component before creating your own custom CSS class. Don't be afraid to experiment with Lightning Design System classes and variants in base components. For more information, see Lightning Design System.

Note: Interactive examples for base components are available in the Component Library.

Migrate Components from the `ui` Namespace

If you're using components in the `ui` namespace, replace them with their `lightning` namespace counterparts.

Note: Components in the `ui` namespace are deprecated as of API version 47.0, the Winter '20 release. We recommend that you use components in the `lightning` namespace instead or use the Lightning web component equivalent. You can continue to use the `ui` components beyond Summer '21 but Salesforce plans to cease support for them in Summer '21. For more information, see Working with Base Lightning Components on page 102.

For migration recommendations for each `ui` component, refer to the Component Library. For example, the ui:button specification notes that you can use `lightning:button`, `lightning:buttonIcon`, or `lightning:buttonIconStateful` instead, depending on your use case. To view examples and usage guidelines for `lightning` namespace components, refer to the Component Library.

The following tables list `ui` components and their recommended counterparts in the `lightning` namespace.

Complex, Interactive Components

The following components contain one or more subcomponents and are interactive.

Type	`ui` Component (Deprecated)	Description	`lightning` Component
Message	`ui:message`	A message notification of varying severity levels	`lightning:notificationsLibrary`
Menu	`ui:menu`	A dropdown list with a trigger that controls its visibility	`lightning:buttonMenu`
	`ui:menuList`	A list of menu items	
	`ui:actionMenuItem`	A menu item that triggers an action	`lightning:menuItem`
	`ui:checkboxMenuItem`	A menu item that supports multiple selection and can be used to trigger an action	
	`ui:radioMenuItem`	A menu item that supports single selection and can be used to trigger an action	
	`ui:menuItemSeparator`	A visual separator for menu items	`lightning:menuDivider`
	`ui:menuItem`	An abstract and extensible component for menu items in a `ui:menuList` component	`lightning:menuItem`
	`ui:menuTrigger`	A trigger that expands and collapses a menu	`lightning:buttonMenu`
	`ui:menuTriggerLink`	A link that triggers a dropdown menu. This component extends `ui:menuTrigger`	

Input Control Components

The following components are interactive, for example, like buttons and checkboxes.

Type	Key Components (Deprecated)	Description	lightning Component
Button	ui:button	An actionable button that can be pressed or clicked	lightning:button, lightning:buttonIcon, or lightning:buttonIconStateful
Checkbox	ui:inputCheckbox	A selectable option that supports multiple selections	lightning:input with checkbox, toggle, or checkbox-button type
	ui:outputCheckbox	Displays a read-only value of the checkbox	lightning:input with checkbox type and readonly attribute
Radio button	ui:inputRadio	A selectable option that supports only a single selection	lightning:input with radio type or lightning:radioGroup
Dropdown List	ui:inputSelect	A dropdown list with options	lightning:combobox
	ui:inputSelectOption	An option in a ui:inputSelect component	

Visual Components

The following components provide informative cues, for example, like error messages and loading spinners.

Type	Key Components (Deprecated)	Description	lightning Component
Field-level error	ui:inputDefaultError	An error message that is displayed when an error occurs	lightning:input with field validation
Spinner	ui:spinner	A loading spinner	lightning:spinner

Field Components

The following components enable you to enter or display values.

Type	Key Components (Deprecated)	Description	lightning Component
Currency	ui:inputCurrency	An input field for entering currency	lightning:input with number type and currency formatter

Type	Key Components (Deprecated)	Description	`lightning` **Component**
	`ui:outputCurrency`	Displays currency in a default or specified format	`lightning:formattedNumber` with `currency` style
Email	`ui:inputEmail`	An input field for entering an email address	`lightning:input` with `email` type
	`ui:outputEmail`	Displays a clickable email address	`lightning:formattedEmail`
Date and time	`ui:inputDate`	An input field for entering a date	`lightning:input` with `date` type
	`ui:inputDateTime`	An input field for entering a date and time	`lightning:input` with `datetime` type
	`ui:outputDate`	Displays a date in the default or specified format	`lightning:formattedDateTime`
	`ui:outputDateTime`	Displays a date and time in the default or specified format	`lightning:formattedDateTime` or `lightning:formattedTime`
Password	`ui:inputSecret`	An input field for entering secret text	`lightning:input` with `password` type
Phone Number	`ui:inputPhone`	An input field for entering a phone number	`lightning:input` with `phone` type
	`ui:outputPhone`	Displays a phone number	`lightning:formattedPhone`
Number	`ui:inputNumber`	An input field for entering a numerical value	`lightning:input` with `number` type
	`ui:outputNumber`	Displays a number	`lightning:formattedNumber`
Range	`ui:inputRange`	An input field for entering a value within a range	`lightning:slider`
Rich Text	`ui:inputRichText`	An input field for entering rich text	`lightning:inputRichText`
	`ui:outputRichText`	Displays rich text	`lightning:formattedRichText`
Text	`ui:inputText`	An input field for entering a single line of text	`lightning:input`
	`ui:outputText`	Displays text	`lightning:formattedText`
Text Area	`ui:inputTextArea`	An input field for entering multiple lines of text	`lightning:textarea`
	`ui:outputTextArea`	Displays a read-only text area	`lightning:formattedText`
URL	`ui:inputURL`	An input field for entering a URL	`lightning:input` with `url` type
	`ui:outputURL`	Displays a clickable URL	`lightning:formattedUrl`

Supporting Accessibility

When customizing components, be careful to preserve code that ensures accessibility, such as the `aria` attributes.

Accessible software and assistive technology enable users with disabilities to use and interact with the products you build. Aura components are created according to W3C specifications so that they work with common assistive technologies. We recommend that you follow the WCAG Guidelines for accessibility when developing with the Lightning Component framework.

IN THIS SECTION:

Accessibility for Base Lightning Components

This section explains the accessibility features on components in the `lightning` namespace, which are also known as base Lightning components.

IN THIS SECTION:

Button Labels

Buttons can appear with text only, an icon and text, or an icon only. To create an accessible button, use the `lightning:button` or `lightning:buttonIcon` base components and set a textual label using the `label` attribute.

Button with text only:

```
<lightning:button label="Search"
                  onclick="{!c.doSomething}"/>
```

Button with icon and text:

```
<lightning:button label="Download" iconName="utility:download"
                  onclick="{!c.doSomething}"/>
```

`lightning:button` implements the button blueprint in the Salesforce Lightning Design System (SLDS) and follows its accessibility guidelines.

Button with icon only:

```
<lightning:buttonIcon iconName="utility:settings"
                      onclick="{!c.doSomething}
                      alternativeText="Settings"/>
```

The `alternativeText` attribute provides a text label that's hidden from view and available to assistive technology.

`lightning:buttonIcon` implements the button icon blueprint in the SLDS and follows its accessibility guidelines.

This example shows the HTML generated by `lightning:buttonIcon`:

```
<!-- Use assistive text to hide the label visually, but show it to screen readers -->
<button>
    ::before
    <span class="slds-assistive-text">Settings</span>
</button>
```

To support tooltip on desktop, include the `title` attribute in addition to the `label` or `alternativeText` attribute. The `title` attribute can be problematic for touch-only devices, keyboard navigation, and assistive technologies. Therefore, it must be used together with `label` or `alternativeText`.

Note: Most ARIA states and properties are supported on these base components. For more information, see the reference documentation in the Component Library.

Other button-based components include:

lightning:buttonGroup
A group of buttons. This component implements the button group blueprint in the SLDS and follows its accessibility guidelines.

lightning:buttonIconStateful
A button with an icon only that retains state. This component implements the button icon blueprint in the SLDS and follows its accessibility guidelines.

lightning:buttonMenu
A dropdown menu with a list of actions or items. This component implements the menu blueprint in the SLDS and follows its accessibility guidelines.

lightning:buttonStateful
A button that retains state. This component implements the button blueprint in the SLDS and follows its accessibility guidelines.

SEE ALSO:

Using Images and Icons

Creating a Form

Component Library: lightning:button documentation

Audio Messages

To convey audio notifications, create a toast using `lightning:notificationsLibrary`. The toast is rendered with `role="alert"`, which enables screen readers to announce the text inside the toast without any additional action by the user.

If you're creating your own feedback mechanism and work with multiple toasts, consider using `role="status"` to persist the toast in the queue. This role reduces the risk of a user missing a toast message. Contrastingly, `role="alert"` overrides previous toasts in the screen reader's speech queue. For more information, see the toast accessibility guideline.

```
<lightning:notificationsLibrary aura:id="notifLib"/>
<lightning:button name="toast" label="Show Toast" onclick="{!c.handleShowToast}"/>
```

```
({
    handleShowToast : function(component, event, helper) {
        component.find('notifLib').showToast({
            "title": "Success!",
            "message": "The record has been updated successfully."
        });
    }
})
```

Alternatively, create a prompt notice to alert a user of system-related issues or updates. The notice is rendered as a modal dialog with `role="dialog"`, and must be dismissed before you can return to the rest of the page.

```
<lightning:notificationsLibrary aura:id="notifLib"/>
<lightning:button name="notice" label="Show Notice" onclick="{!c.handleShowNotice}"/>
```

```
({
    handleShowNotice : function(component, event, helper) {
        component.find('notifLib').showNotice({
            "variant": "error",
            "header": "Something has gone wrong!",
            "message": "Unfortunately, there was a problem updating the record.",
            closeCallback: function() {
                alert('You closed the alert!');
            }
        });
    }
})
```

`lightning:notificationsLibrary` implements the prompt and toast blueprint in the Salesforce Lightning Design System and follows its accessibility guidelines.

SEE ALSO:

Component Library: lightning:notificationsLibrary documentation

Forms, Fields, and Labels

Input components are designed to make it easy to assign labels to form fields. Labels build a programmatic relationship between a form field and its textual label. When using a placeholder in an input component, set the `label` attribute for accessibility.

Use `lightning:input` to create accessible input fields and forms. You can use `lightning:textarea` in preference to the `<textarea>` tag for multi-line text input or `lightning:select` instead of the `<select>` tag.

```
<lightning:input name="myInput" label="Search" />
```

If your code fails, check the label element during component rendering. The label element's `for` attribute must match the value of the input control's `id` attribute. Alternatively, wrap the label around an input. Input controls include `<input>`, `<textarea>`, and `<select>`.

Here's an example of the HTML generated by `lightning:input`.

```
<!-- Use label/for -->
<label for="fullname">Enter your full name:</label>
<input type="text" id="fullname" />

<!-- Use an implicit label -->
<label>Enter your full name:
    <input type="text" id="fullname"/>
</label>
```

SEE ALSO:

Using Labels

Creating a Form

Component Library: lightning:input documentation

Component Library: lightning:textarea documentation

Using Images and Icons

To display images, use the HTML `<img>` element. Include an image in your component by uploading it as a static resource on page 58 or content asset on page 55.To display an icon, use the `lightning:icon` component, which gives you access to Salesforce Lightning Design System icons or your own custom icon. To display an avatar, use `lightning:avatar`.Consider `lightning:buttonIcon` or `lightning:buttonIconStateful` to display an actionable image such as a Like or Follow image.

Follow these accessibility guidelines when using images and icons in your component.

Informational Images and Icons

Informational images and icons can provide information that's not available in the text, such as an image that represents an approved step. Include alternative text for images and icons to help users without access to the images and icons. Use the `alt` attribute with the `<img>` element and `alternativeText` attribute with the base Lightning components.

Image with Alternative Text:

```
<img src="{!$Resource.profile_pic}" alt="User avatar"/>
```

Icon with Alternative Text:

```
<lightning:icon iconName="action:approval" size="large" alternativeText="Indicates
approval"/>
```

Avatar with Alternative Text:

```
<lightning:avatar src="{!$Resource.profile_pic}" alternativeText="John Smith"/>
```

Stateful Button Icon with Alternative Text:

An informational image or icon, such as a Like icon, is actionable and can stand alone in a button or hyperlink.

```
<lightning:buttonIconStateful
        iconName="utility:like"
        selected="{! v.liked }"
        onclick="{! c.handleLikeButtonClick }"
        alternativeText="Like" />
```

CSS with Alternative Text:

If you use CSS to display an informational image, you must provide assistive text.

```
<div class="Following">
    <span class="slds-assistive-text">Following</span>
</div>
```

Decorative Images and Icons

Decorative images are images that can be removed without affecting the logic or content of the page. Assistive text is optional for decorative images.

For example, placing an add icon or checkmark icon next to a text label reinforces the text's meaning but adds no new information. Consider this Follow button with an add icon next to it. When clicked, the button label changes to "Following" and its icon updates to a checkmark. The icons don't require assistive text.

```
<lightning:buttonStateful
        labelWhenOff="Follow"
        labelWhenOn="Following"
        iconNameWhenOff="utility:add"
        iconNameWhenOn="utility:check"
        state="{! v.buttonstate }"
        onclick="{! c.handleClick }"
    />
```

The base Lightning components discussed in this topic implement the iconography design and accessibility guidelines in the Salesforce Lightning Design System.

SEE ALSO:

Component Library: lightning:avatar documentation

Component Library: lightning:buttonIcon documentation

Component Library: lightning:buttonIconStateful documentation

Component Library: lightning:buttonStateful documentation

Component Library: lightning:icon documentation

Events

Although you can attach an `onclick` event to any type of element, for accessibility, consider only applying this event to elements that are actionable in HTML by default, such as `<a>`, `<button>`, or `<input>` tags in component markup. You can use an `onclick` event on a `<div>` tag to prevent event bubbling of a click.

Menus

A menu is a dropdown list with a trigger that controls the visibility of the list items. To create an accessible menu, use `lightning:buttonMenu`. Provide a text label or assistive text, and specify a list of menu items using `lightning:menuItem`. The dropdown menu items are hidden by default.

This example creates a menu with several items:

```
<lightning:buttonMenu iconName="utility:settings"
                      alternativeText="Settings"
                      onselect="{! c.handleMenuSelect }">
    <lightning:menuItem label="Font" value="font" />
    <lightning:menuItem label="Size" value="size"/>
    <lightning:menuItem label="Format" value="format" />
</lightning:buttonMenu>
```

The `alternativeText` attribute provides a text label that's hidden from view and available to assistive technology.

`lightning:buttonMenu` implements the menu blueprint in the Salesforce Lightning Design System and follows its accessibility guidelines.

SEE ALSO:

Component Library: lightning:buttonMenu documentation

Write Aura Component Accessibility Tests

When you develop with Aura components, you can use Salesforce's test tools to check for common accessibility issues.

You can call Aura accessibility tests in two environments.

- For JavaScript tests, use `$A.test.assertAccessible()`.
- For WebDriver tests, use `auraTestingUtil.assertAccessible()`.

These functions check the rendered DOM elements to make sure they pass Salesforce's accessibility validation.

When you use these tools, there are two outcomes: pass or fail. If the tool doesn't find any accessibility exceptions, it returns an empty string. If the tool does find accessibility exceptions, it returns the accessibility rule that failed, the erroneous tag, and a stack trace of where it was found in the code.

Since Aura components and pages are dynamic, make sure to retest your components' accessibility every time something changes in the DOM. Otherwise you aren't checking every UI state your users encounter.

The Aura accessibility tests look for these issues:

- Images without `alt` attributes
- Anchor elements without textual content
- `input` elements without associated labels
- Radio button groups not in `fieldset` tags
- `iframe` or `frame` elements with empty `title` attributes
- `fieldset` elements without `legend` tags
- `th` element without a `scope` attribute
- `head` element with an empty `title` attribute
- Headings (`H1`, `H2`, and so forth) increasing by more than one level at a time
- CSS color contrast ratio between text and background less than 4.5:1

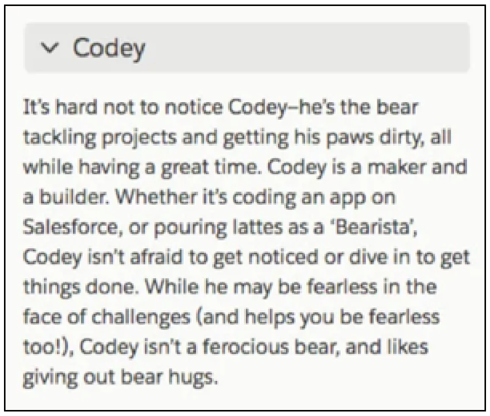
These tests aren't all-encompassing. If your code passes every test, it's not a guarantee that your product is fully accessible. However, these tests do surface major accessibility issues, and ensure that your code remains accessible.

IN THIS SECTION:

Accessibility Tests Example

If you've made a component accessible, write tests to make sure it stays that way. You can write automated tests for a variety of accessibility concerns, including expected keyboard functionality and that the role, state, and property ARIA values for HTML elements are correct.

Let's look at an example that tests an expandable section. When you click Codey's name, the section expands to tell you more about him, and when you click his name again, the section collapses.



It's hard not to notice Codey–he's the bear tackling projects and getting his paws dirty, all while having a great time. Codey is a maker and a builder. Whether it's coding an app on Salesforce, or pouring lattes as a 'Bearista', Codey isn't afraid to get noticed or dive in to get things done. While he may be fearless in the face of challenges (and helps you be fearless too!), Codey isn't a ferocious bear, and likes giving out bear hugs.

Here's some pseudocode for an Aura component test that toggles the collapsed and expanded state of an expandable section.

```
testToggleExpandCollapse : {
  test : [
    function(cmp) {
      // Default: collapsed
      this.assertCollapsed(cmp);
      // Toggle to expanded
      this.clickToggleButton(cmp);
      this.assertExpanded(cmp);
      // Toggle back to collapsed
      this.clickToggleButton(cmp);
      this.assertCollapsed(cmp);
    }
  ]
}
```

First, we assert the element is collapsed by default, then we click the toggle button, verify it's expanded, click the toggle button again, and verify it's collapsed.

How can we embed accessibility checks into this test? Let's explore the two helper functions `assertCollapsed` and `assertExpanded`.

```
assertCollapsed : (cmp) {
  var button = this.getButton(cmp);
  var section = this.getSection(cmp);
  // Button indicates section is collapsed
  aura.test.assertEquals(
    button.getAttribute('aria-expanded'),
    "false",
    "Button should indicate it's collapsed"
  );
  // Section is visually closed
  aura.test.assertFalse(
    section.classname.indexOf('slds-is-open') > -1,
    "Section should be collapsed"
  );
}
```

For an expandable section to be accessible, it must communicate its expanded or collapsed state to assistive technology users, such as screen reader users. The best way to make the section accessible is with an ARIA state attribute, `aria-expanded`, on the button, which is `true` when the section is expanded and `false` otherwise. To make sure that this attribute is always properly set, we can assert it has the correct value. In `assertCollapsed`, we assert that `aria-expanded` has a value of `false`. Now, in `assertExpanded`, we can assert that aria-expanded has a value of `true`.

```
assertExpanded : (cmp) {
  var button = this.getButton(cmp);
  var section = this.getSection(cmp);
  // Button indicates section is expanded
  aura.test.assertEquals(
    button.getAttribute('aria-expanded'),
    "true",
    "Button should indicate it's expanded"
  );
  // Section is visually open
  aura.test.assertTrue(
    section.classname.indexOf('slds-is-open') > -1,
    "Section should be open"
  );
}
```

If the code that's setting `aria-expanded` regresses, we catch the bug before it reaches screen reader users. Now let's go back to our `testToggleExpandCollapse` test case. Let's add `$A.test.assertAccessible()` in two strategic places so that we run Salesforce's default set of accessibility checks against the section's expanded and collapsed states. Remember, we want to test every state, not just one. If we test only the collapsed state, we might miss accessibility bugs in the expanded section.

```
testToggleExpandCollapse : {
  test : [
    function(cmp) {
      // Default: collapsed
      this.assertCollapsed(cmp);
      // Toggle to expanded
      this.clickToggleButton(cmp);
      this.assertExpanded(cmp);
      $A.test.assertAccessible();
```

```
        // Toggle back to collapsed
        this.clickToggleButton(cmp);
        this.assertCollapsed(cmp);
        $A.test.assertAccessible();
      }
    ]
}
```

Now we have accessibility checks running automatically in our custom tests.

Other Accessibility Automation Tools

There are a number of robust open-source tools and mobile test frameworks for testing for accessibility.

Open-Source Tools

- axe
- Linters (es-lint and others)
  - https://github.com/evcohen/eslint-plugin-jsx-a11y
  - https://github.com/reactjs/react-a11y

Mobile Test Frameworks

- *iOS Documentation*: About Accessibility Verification on iOS
- *Android Documentation*: Test Your App's Accessibility

Writing Documentation for the Component Library

Documentation helps developers use your components to develop their apps more effectively. You can provide interactive examples, documentation, and specification descriptions for a component, event, or interface.

Each component, event, or interface has a root definition that defines the element's metadata, as well as attributes, events, or methods.

Component

A component's root definition is specified in the `<aura:component>` tag contained in `componentName.cmp`, as described in Component Markup.

Event

An event's root definition is specified in the `<aura:event>` tag contained in `componentEvent.evt`, as described in Create Custom Component Events.

Interface

An interface's root definition is specified in the `<aura:interface>` tag contained in the `interfaceName.intf`, as described in Interfaces.

The root definition tag determines whether the element is exposed in the Component Library. You provide the documentation for each element in an `.auradoc` file that accompanies the other files that define the component, event, or interface.

Viewing the Documentation

View the Component Library through your org at `https://*MyDomainName*.my.salesforce.com/docs/component-library`. Alternatively, view the unauthenticated Component Library at `https://developer.salesforce.com/docs/component-library/`.

For namespaces that you own, elements with either `access="global"` or `access="public"` (default) are surfaced in the Component Library when it's accessed through your org.

In the unauthenticated Component Library, only the elements with `access="global"` are visible.

For managed package namespaces, only elements with `access="global"` are surfaced in the Component Library. Elements with `access="public"` can be used only by components in the same namespace in the same org. They aren't available to other orgs that install the package, so they aren't surfaced in the Component Library. Only global components in managed packages are visible because they are intended for use in any namespace in any org.

Each element can display up to three tabs in the following order.

Example

Displays interactive examples denoted by the `<aura:example>` tag in the `.auradoc` file. This tab is hidden if no examples are wired up in your `.auradoc` file. This tab is not supported if your component has dependency on org data, such as with `lightning:recordForm`.

Documentation

Displays the content of the `.auradoc` file. This tab is hidden if an `.auradoc` file is not available for your component, event, or interface.

Specification

Displays the description of the root definition, attributes, and methods. For namespaces you own, attributes and methods with either `access="global"` or `access="public"` (default) are surfaced in the Component Library when it's accessed through your org. For managed package namespaces, only `access="global"` attributes and methods are visible.

IN THIS SECTION:

Creating Examples

Examples are interactive and help others learn about a component, event, or interface.

Creating Documentation Content

Documentation provides usage guidelines and code samples about a component, event, or interface.

Providing Specification Information and Descriptions

Descriptions on the **Specification** tab describes a root definition and its attributes and methods.

SEE ALSO:

Controlling Access

Creating Examples

Examples are interactive and help others learn about a component, event, or interface.

Note: You must create an `.auradoc` file before creating an example. For more information, see Creating Documentation Content.

In the Component Library, the **Example** tab renders your example with its code. For instance, see the `lightning:avatar` example at https://developer.salesforce.com/docs/component-library/bundle/lightning:avatar/. Each component, event, or interface can have multiple examples to demonstrate different use cases.

The following is an example component that demonstrates how to use `lightning:avatar`. The example uses a component in the `lightningcomponentdemo` namespace. You can create an example component in your own namespace, such as the default `c` namespace.

The example component is rendered as an interactive demo in the **Example** tab when it's wired up using the `<aura:example>` tag in the `.auradoc` file.

```
<aura:documentation>
    <aura:description>
        <!-- Your content here -->
    </aura:description>

   <aura:example name="exampleAvatarBasic" ref="lightningcomponentdemo:exampleAvatarBasic"
 label="Basic Avatar">
        The following example creates an avatar with the default size and variant.
        The initials "BW" is displayed if the image path denoted by the src attribute
        is invalid or fails to load for any reason, such as when the user is offline.
    </aura:example>
</aura:documentation>
```

The text content within the `<aura:example>` tag is rendered as a tooltip and as a subtitle below the label text. Any HTML markup is removed.

Examples are not supported for components with dependency on org data, such as with `lightning:recordEditForm`, `lightning:recordForm`, and `lightning:recordViewForm`. Examples are also not supported for components that import internal JavaScript libraries, which include:

- `lightning:formattedAddress`
- `lightning:formattedRichText`
- `lightning:inputAddress`
- `lightning:inputName`
- `lightning:inputRichText`

Creating Documentation Content

Documentation provides usage guidelines and code samples about a component, event, or interface.

In the Component Library, the **Documentation** tab renders content from your `.auradoc` file. For an example, see the `lightning:avatar` Documentation tab at https://developer.salesforce.com/docs/component-library/bundle/lightning:avatar/documentation. Each component, event, or interface element can have one `.auradoc` file.

Writing the Documentation

Provide your content in HTML markup. Add usage guidelines and code samples to help developers use your component, event, or interface easily.

To provide documentation, click **DOCUMENTATION** in the component sidebar of the Developer Console, which creates a `.auradoc` file for your component.

A `.auradoc` file contains these tags.

Tag	Description
`<aura:documentation>`	Required. Creates documentation for a component, event, or interface.
`<aura:description>`	Required. Describes the component using HTML markup.
`<aura:example>`	References an example that demonstrates how the component is used. Supports HTML markup, which displays as text preceding the visual output and example component source. Use the example to create an interactive experience that demonstrates features of your component. • `name`: The API name of the example • `ref`: The reference to the example component in the format `<namespace:exampleComponent>` • `label`: The title that describes the example Each `.auradoc` file can contain multiple `<aura:example>` tags.

Basic Formatting

We recommend that you use only the tags listed here and in the following sections. The Component Library strips out or escapes unexpected tags and attributes for security reasons.

Make sure to include closing tags.

Heading

Only <h4> headings are supported for headings.

```
<h4>Usage Considerations</h4>
```

Paragraph

```
<p>Some cool paragraph about a component</p>
```

Code Formatting

```
<p>Here’s a paragraph on the <code>c:myComponentName</code> component.</p>
```

Code Blocks

Create a code block using the `<pre>` tag and an embedded `<code>` tag for code highlighting. Code markup must be escaped. For example, replace < characters with `&lt;`.

```
<pre><code class="language-markup">&lt;aura:component>
   &lt;lightning:accordion activeSectionName="B">
     &lt;lightning:accordionSection name="A" label="Accordion Title A">This is the content
 area for section A&lt;/lightning:accordionSection>
     &lt;lightning:accordionSection name="B" label="Accordion Title B">This is the content
 area for section B&lt;/lightning:accordionSection>
   &lt;/lightning:accordion>
&lt;/aura:component></code></pre>
```

To enable code highlighting in the Component Library, add the `class` attribute to the `<code>` tag. Code samples with Aura markup use the class `language-markup`. JavaScript controllers use `language-js`, and CSS use `language-css`.

Links

The Component Library supports links to other component reference pages, external links, and cross-tab linking from the component reference to the Lightning Web Components Developer Guide.

Anchor links are currently not supported.

Create a link to another component reference page

```
<a href="/docs/component-library/bundle/lightning:input/documentation">
   lightning:input</a>
```

Create an external link

```
<p>
    For more information, see the
    <a
href=""https://developer.salesforce.com/docs/atlas.en-us.api_console.meta/api_console/"

    target="_blank">Console Developer Guide</a>.
</p>
```

Create a link to a topic in the Lightning Web Components Developer Guide

```
<a href="/docs/component-library/documentation/lwc/lwc.use_navigate">
   Navigate to Pages
</a>
```

Lists

Lists present related items and can be bulleted or numbered.

Create a bulleted list

```
<p><code>lightning:listView</code> supports the following features.</p>
<ul>
    <li>Inline editing</li>
    <li>Mass inline editing for single record types</li>
    <li>Resizing and sorting of columns</li>
    <li>Search</li>
    <li>Text wrapping</li>
    <li>Loading of additional rows</li>
</ul>
```

Create a numbered list

```
<p>The toolbar provides menus and buttons that are ordered within the following
categories.</p>
<ol>
    <li><code>FORMAT_FONT</code>: Font family and size menus. The font menu provides
the following font selection: Arial, Courier, Garamond, Salesforce Sans, Tahoma, Times
 New Roman, and Verdana. The font selection defaults to Salesforce Sans with a size of
 12px. Supported font sizes are: 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36,
 48, and 72. When you copy and paste text in the editor, the font is preserved only if
 the font is available in the font menu.</li>
   <li><code>FORMAT_TEXT</code>: Bold, Italic, Underline, and Strikethrough buttons.</li>
```

```
    <li><code>FORMAT_BODY</code>: Bulleted List, Numbered List, Indent, and Outdent
buttons.</li>
    <li><code>ALIGN_TEXT</code>: Left Align Text, Center Align Text, and Right Align
Text buttons.</li>
    <li><code>INSERT_CONTENT</code>: Image button. The Image button displays if you
include the <code>lighting:insertImageButton</code> component in
<code>lightning:inputRichText</code>.</li>
    <li><code>REMOVE_FORMATTING</code>: Remove formatting button, which stands alone at
 the end of the toolbar.</li>
</ol>
```

Tables

Tables are useful for presenting a list of items with several accompanying descriptions. Nesting a bulleted list in a table is currently not supported.

```
<table>
    <tr>
        <th>Property</th>
        <th>Type</th>
        <th>Description</th>
    </tr>
    <tr>
        <td>label</td>
        <td>string</td>
        <td>The text that displays next to a radio button.</td>
    </tr>
    <tr>
        <td>value</td>
        <td>string</td>
        <td>The string that's used to identify which radio button is selected.</td>
    </tr>
</table>
```

Providing Specification Information and Descriptions

Descriptions on the **Specification** tab describes a root definition and its attributes and methods.

In the Component Library, the **Specification** tab renders descriptions from your `.cmp`, `.evt`, or `.intf` file. For instance, see the Specification tab for `lightning:avatar` at https://developer.salesforce.com/docs/component-library/bundle/lightning:avatar/specification.

HTML markup is not supported in inline descriptions.

The specification information is generated based on the root-level tag, which looks like this.

```
<aura:component
    access="global"
    implements="lightning:myInterface"
    minVersion="41.0"
    description="A collection of vertically stacked sections with multiple content areas.
 This component requires version 41.0 and later.">
```

The specification information includes:

Access Level

Only root definitions, attributes, and methods marked with `access="global"` are surfaced in the Component Library.

Abstract

A root definition with `abstract="true"` denotes that it's abstract. An abstract component can't be used directly in markup. The default is `false`.

Extensible

A root definition with `extensible="true"` denotes that it's extensible, which makes it a super component. A sub component that extends a super component inherits the attributes of the super component. The default is `false`.

These tags support inline descriptions via the `description` attribute.

`<aura:component>`

The root definition tag in a component `.cmp` file.

```
<aura:component description="Represents a button element.">
```

`<aura:event>`

The root definition tag in an event `.evt` file.

```
<aura:event type="COMPONENT"
    description="Indicates that a key has been pressed.">
```

`<aura:interface>`

The root definition tag in an interface `.intf` file.

```
<aura:interface name="label"
    type="String"
    description="A common interface for date components.">
```

`<aura:attribute>`

An attribute tag in a component, event, or interface file.

```
<aura:attribute name="label"
    type="String"
    description="The text to be displayed on the button.">
```

`<aura:method>`

A method tag in a component , event, or interface file. Each method tag can contain multiple attribute tags.

```
<aura:method name="setCustomValidity" description="Sets a custom error message.">
  <aura:attribute name="message" type="String"
      description="The string that describes the error. If message is an empty string,
 the error message is reset."/>
</aura:method>
```

CHAPTER 4 Using Components

In this chapter ...

You can use components in many different contexts. This section shows you how.

Important: Customizing Aura components isn't supported in Starter and Pro Suite Editions.

Aura Component Bundle Design Resources

Use a design resource to control which attributes are exposed to builder tools like the Lightning App Builder, Experience Builder, or Flow Builder. A design resource lives in the same folder as your `.cmp` resource, and describes the design-time behavior of the Aura component—information that visual tools need to display the component in a page or app.

For example, here's a simple design resource that goes in a bundle with a "Hello World" component. We'll build on this example as we move through the supported tags and attributes.

```
<design:component label="Hello World">
    <design:attribute name="subject" label="Subject" description="Name of the person you
want to greet" />
    <design:attribute name="greeting" label="Greeting" />
</design:component>
```

`design:component`

This is the root element for the design resource. It contains the component's design-time configuration for tools such as the App Builder to use.

Attribute	Description
label	Sets the label of the component when it displays in tools such as App Builder. When creating a custom Lightning page template component, this text displays as the name of the template in the Lightning App Builder new page wizard.

Note: Label expressions in markup are supported in `.cmp` and `.app` resources only.

`design:attribute`

To make an Aura component attribute available for admins to edit in tools such as the App Builder, add a `design:attribute` node for the attribute into the design resource. An attribute marked as required in the component definition automatically appears, unless it has a default value assigned to it.

For Lightning page interfaces, the design resource supports only attributes of type `Integer`, `String`, or `Boolean`. To see which attribute types the `lightning:availableForFlowScreens` interface supports, go to Which Custom Lightning Component Attribute Types Are Supported in Flows?.

Note: In a `design:attribute` node, Flow Builder supports only the `name`, `label`, `description`, and `default` attributes. The other attributes, like `min` and `max`, are ignored.

Attribute	Description
datasource	Renders a field as a picklist, with static values. Only supported for `String` attributes. `<design:attribute name="Name" datasource="value1,value2,value3" />` You can also set the picklist values dynamically using an Apex class. See Create Dynamic Picklists for Your Custom Components on page 189 for more information.

Attribute	Description
	Any `String` attribute with a `datasource` in a design resource is treated as a picklist.
default	Sets a default value on an attribute in a design resource. `<design:attribute name="Name" datasource="value1,value2,value3" default="value1" />`
description	Displays as an i-bubble for the attribute in the tool.
label	Attribute label that displays in the tool.
max	If the attribute is an `Integer`, this sets its maximum allowed value. If the attribute is a `String`, this is the maximum length allowed.
min	If the attribute is an `Integer`, this sets its minimum allowed value. If the attribute is a `String`, this is the minimum length allowed.
name	Required attribute. Its value must match the `aura:attribute` name value in the `.cmp` resource.
placeholder	Input placeholder text for the attribute when it displays in the tool.
required	Denotes whether the attribute is required. If omitted, defaults to `false`.
type	The design attribute's data type. `Color` is the only valid value. The `Color` type displays a color picker in Experience Builder. Applies only to components that implement the `forceCommunity:availableForAllPageTypes` interface. Supported only for `aura:attribute` elements of type `String` in the `.cmp` resource. Use the `default` attribute to specify RGBA, RGB, or hex strings. For example: `<design:attribute type="Color" name="buttonColor" default="rgba(0, 255, 255, 1)" />`

Note: Label expressions in markup are supported in `.cmp` and `.app` resources only.

`<design:suppportedFormFactors>` and `<design:suppportedFormFactor>`

Use these tag sets to designate which devices your component supports. The `design:suppportedFormFactor` subtag supports the `type` attribute. Valid `type` values are `Large` (desktop) and `Small` (phone).

If you don't declare form factor support for a component, then by default, it supports the same form factors as the page types that it's assigned to. App and record pages support the `Large` and `Small` form factors. Home pages support only the `Large` form factor.

Components on app and record pages can render on both mobile and desktop because those pages support both phone and desktop. Components on Home pages can render only on desktop because Home pages are supported only for desktop.

If you have an app or record page—which support both desktop and phone—you can use `design:suppportedFormFactor` to configure a component to render only when the page is viewed on a particular device. For example, if you restrict form factor support for your app page component to `Small`, the app page drops the component when the page is viewed on desktop. The app page displays the component when the page is viewed on a phone.

Here's the "Hello World" component design resource, with both desktop and phone support added.

```
<design:component label="Hello World">
    <design:attribute name="subject" label="Subject" description="Name of the person you
want to greet" />
    <design:attribute name="greeting" label="Greeting" />
    <design:supportedFormFactors>
        <design:supportedFormFactor type="Large"/>
        <design:supportedFormFactor type="Small"/>
    </design:supportedFormFactors>
</design:component>
```

You can add this tag set to your component design file to create custom page templates that support only desktop, only phone, or both.

`<sfdc:objects>` and `<sfdc:object>`

Use these tag sets to restrict your component to one or more objects.

Note: `<sfdc:objects>` and `<sfdc:object>` aren't supported in Experience Builder or the Flow Builder. They're also ignored when setting a component to use as an object-specific action or to override a standard action.

Here's the same "Hello World" component's design resource restricted to two objects.

```
<design:component label="Hello World">
    <design:attribute name="subject" label="Subject" description="Name of the person you
want to greet" />
    <design:attribute name="greeting" label="Greeting" />
    <design:supportedFormFactors>
        <design:supportedFormFactor type="Large"/>
        <design:supportedFormFactor type="Small"/>
    </design:supportedFormFactors>
    <sfdc:objects>
        <sfdc:object>Custom__c</sfdc:object>
        <sfdc:object>Opportunity</sfdc:object>
    </sfdc:objects>
</design:component>
```

If an object is installed from a package, add the ***`namespace`***`__` string to the beginning of the object name when including it in the `<sfdc:object>` tag set. For example: `objectNamespace__ObjectApiName__c`.

These tag sets don't support external objects.

See the User Interface API Developer Guide for the list of supported objects.

SEE ALSO:

Configure Components for Lightning Pages and the Lightning App Builder

Tips and Considerations for Configuring Components for Lightning Pages and the Lightning App Builder

Use Aura Components in Lightning Experience and the Salesforce Mobile App

Customize and extend Lightning Experience and the Salesforce mobile app with Aura components. Launch components from tabs, apps, and actions.

IN THIS SECTION:

Configure Components for Custom Tabs

Add the `force:appHostable` interface to an Aura component to allow it to be used as a custom tab in Lightning Experience, the Salesforce mobile app, and Salesforce mobile web.

Add Aura Components as Custom Tabs in a Lightning Experience App

Make your Aura components available for Lightning Experience users on desktop and in the Salesforce mobile app by displaying them in a custom tab in a Lightning Experience app.

Configure Components for Custom Actions

Add the `force:lightningQuickAction` or `force:lightningQuickActionWithoutHeader` interface to an Aura component to enable it to be used as a custom action in Lightning Experience or the Salesforce mobile app. You can use components that implement one of these interfaces as object-specific or global actions in both Lightning Experience and the Salesforce mobile app.

Configure Components for Record-Specific Actions

Add the `force:hasRecordId` interface to an Aura component to enable the component to be assigned the ID of the current record. The current record ID is useful if the component is used on a Lightning record page, as an object-specific custom action or action override in Lightning Experience or the Salesforce app, and so on.

Create an Email as a Quick Action

In a custom component, create a button to launch the email composer with pre-populated content. To launch a record create a page with pre-populated field values, use the `lightning:pageReferenceUtils` and `lightning:navigation` components together.

Override Standard Actions with Aura Components

Add the `lightning:actionOverride` interface to an Aura component to enable the component to be used to override a standard action on an object. You can override the View, New, Edit, and Tab standard actions on most standard and all custom components. Overriding standard actions allows you to customize your org using Lightning components, including completely customizing the way you view, create, and edit records.

Configure Components for Custom Tabs

Add the `force:appHostable` interface to an Aura component to allow it to be used as a custom tab in Lightning Experience, the Salesforce mobile app, and Salesforce mobile web.

Components that implement this interface can be used to create tabs in both Lightning Experience, the Salesforce mobile app, and the Salesforce mobile web.

Example: **Example Component**

```
<!--simpleTab.cmp-->
<aura:component implements="force:appHostable">

    <!-- Simple tab content -->
    <h1>Lightning Component Tab</h1>

</aura:component>
```

The `appHostable` interface makes the component available for use as a custom tab. It doesn't require you to add anything else to the component.

SEE ALSO:

Add Aura Components as Custom Tabs in a Lightning Experience App

Add Aura Components as Custom Tabs in a Lightning Experience App

Make your Aura components available for Lightning Experience users on desktop and in the Salesforce mobile app by displaying them in a custom tab in a Lightning Experience app.

Before you begin, ensure that your component is configured for custom tab usage. See Configure Components for Custom Tabs.

Follow these steps to include your component in a Lightning Experience app and make it available to desktop and mobile users in your org.

1. Create a custom tab for the component.
   a. From Setup, enter *`Tabs`* in the Quick Find box, then select **Tabs**.
   b. Click **New** in the Lightning Component Tabs related list.
   c. Select the Lightning component that you want to make available to users.
   d. Enter a label to display on the tab.
   e. Select the tab style and click **Next**.
   f. When prompted to add the tab to profiles, accept the default and click **Save**.

      Your Lightning component is now available from the All Items section of the App Launcher on desktop, and the All Items navigation menu item in the Salesforce mobile app.

2. Add your Lightning components to a Lightning app's navigation.
   a. From Setup, enter *`Apps`* in the Quick Find box, then select **App Manager**.
   b. Edit an existing app or create a new app.
   c. On the Navigation Items screen, select your Lightning component tab from the Available Items list and move it to the Selected Items list.
   d. Save the app.

3. To check your output, navigate to the App Launcher in Lightning Experience on desktop or in the Salesforce mobile app. Select the custom app to see the components that you added.

Configure Components for Custom Actions

Add the `force:lightningQuickAction` or `force:lightningQuickActionWithoutHeader` interface to an Aura component to enable it to be used as a custom action in Lightning Experience or the Salesforce mobile app. You can use components that implement one of these interfaces as object-specific or global actions in both Lightning Experience and the Salesforce mobile app.

When used as actions, components that implement the `force:lightningQuickAction` interface display in a panel with standard action controls, such as a **Cancel** button. These components can display and implement their own controls in the body of the panel, but can't affect the standard controls. It should nevertheless be prepared to handle events from the standard controls.

If instead you want complete control over the user interface, use the `force:lightningQuickActionWithoutHeader` interface. Components that implement `force:lightningQuickActionWithoutHeader` display in a panel without additional controls and are expected to provide a complete user interface for the action.

These interfaces are mutually exclusive. That is, components can implement either the `force:lightningQuickAction` interface or the `force:lightningQuickActionWithoutHeader` interface, but not both. This should make sense; a component can't both present standard user interface elements and *not* present standard user interface elements.

Note: For your Aura component to work as a custom action, you must set a default value for each component attribute marked as required.

Example: **Example Component**

Here's an example of a component that can be used for a custom action, which you can name whatever you want—perhaps "Quick Add". (A component and an action that uses it don't need to have matching names.) This component quickly adds two numbers together.

```
<!--quickAdd.cmp-->
<aura:component implements="force:lightningQuickAction">

    <!-- Very simple addition -->

    <lightning:input type="number" name="myNumber" aura:id="num1" label="Number 1"/>
+
    <lightning:input type="number" name="myNumber" aura:id="num2" label="Number 2"/>

    <br/>
    <lightning:button label="Add" onclick="{!c.clickAdd}"/>

</aura:component>
```

The component markup simply presents two input fields, and an **Add** button.

The component's controller does all the real work.

```
/*quickAddController.js*/
({
    clickAdd: function(component, event, helper) {

        // Get the values from the form
        var n1 = component.find("num1").get("v.value");
        var n2 = component.find("num2").get("v.value");

        // Display the total in a "toast" status message
        var resultsToast = $A.get("e.force:showToast");
        resultsToast.setParams({
            "title": "Quick Add: " + n1 + " + " + n2,
            "message": "The total is: " + (n1 + n2) + "."
        });
        resultsToast.fire();

        // Close the action panel
        var dismissActionPanel = $A.get("e.force:closeQuickAction");
        dismissActionPanel.fire();
    }

})
```

Retrieving the two numbers entered by the user is straightforward, though a more robust component would check for valid inputs, and so on. The interesting part of this example is what happens to the numbers and how the custom action resolves.

The results of the add calculation are displayed in a "toast," which is a status message that appears at the top of the page. The toast is created by firing the `force:showToast` event. A toast isn't the only way you could display the results, nor are actions the only use for toasts. It's just a handy way to show a message at the top of the screen in Lightning Experience or the Salesforce mobile app.

What's interesting about using a toast here, though, is what happens afterward. The `clickAdd` controller action fires the `force:closeQuickAction` event, which dismisses the action panel. But, even though the action panel is closed, the toast still displays. The `force:closeQuickAction` event is handled by the action panel, which closes. The `force:showToast` event is handled by the `one.app` container, so it doesn't need the panel to work.

SEE ALSO:

Configure Components for Record-Specific Actions

Configure Components for Record-Specific Actions

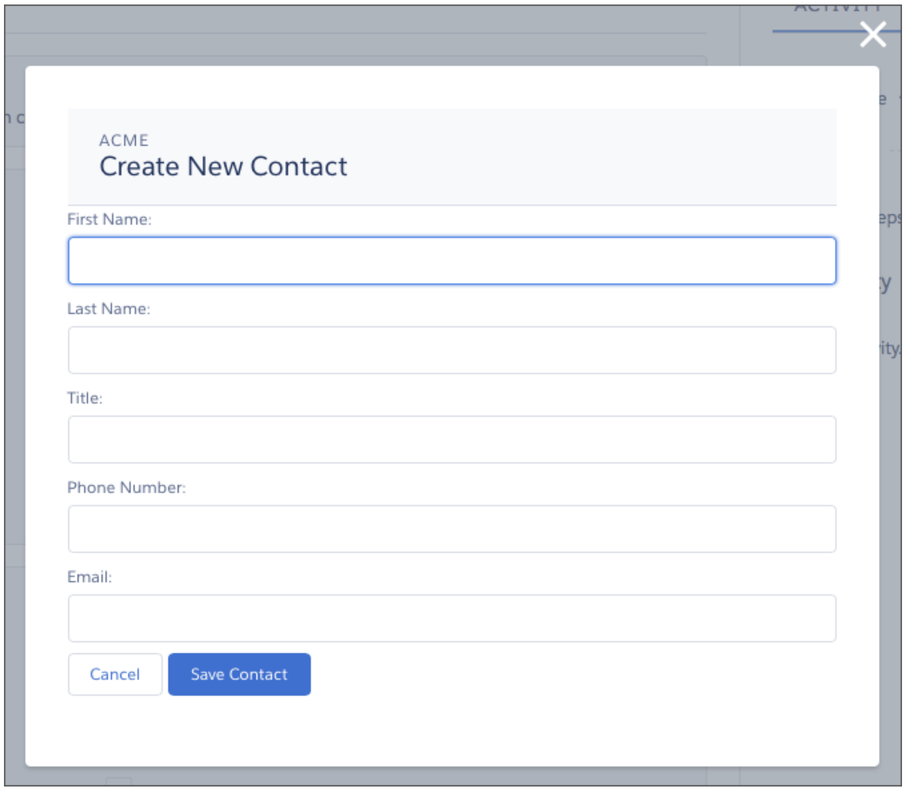
Add the `force:hasRecordId` interface to an Aura component to enable the component to be assigned the ID of the current record. The current record ID is useful if the component is used on a Lightning record page, as an object-specific custom action or action override in Lightning Experience or the Salesforce app, and so on.

`force:hasRecordId` is a *marker interface*. A marker interface is a signal to the component's container to add the interface's behavior to the component.

The `recordId` attribute is set only when you place or invoke the component in an explicit record context. For example, when you place the component directly on a record page layout, or invoke it as an object-specific action from a record page or object home. In all other cases, such as when you invoke the component as a global action, or create the component programmatically inside another component, `recordId` isn't set, and your component shouldn't depend on it.

Example: **Example of a Component for a Record-Specific Action**

This extended example shows a component designed to be invoked as a custom object-specific action from the detail page of an account record. After creating the component, you need to create the custom action on the account object, and then add the action to an account page layout. When opened using an action, the component appears in an action panel that looks like this:



The component definition begins by implementing both the `force:lightningQuickActionWithoutHeader` and the `force:hasRecordId` interfaces. The first makes it available for use as an action and prevents the standard controls from displaying. The second adds the interface's automatic record ID attribute and value assignment behavior, when the component is invoked in a record context.

`quickContact.cmp`

```
<aura:component controller="QuickContactController"
    implements="force:lightningQuickActionWithoutHeader,force:hasRecordId">

    <aura:attribute name="account" type="Account" />
    <aura:attribute name="newContact" type="Contact"
        default="{ 'sobjectType': 'Contact' }" /> <!-- default to empty record -->

    <aura:handler name="init" value="{!this}" action="{!c.doInit}" />

    <!-- Display a header with details about the account -->
    <div class="slds-page-header" role="banner">
        <p class="slds-text-heading_label">{!v.account.Name}</p>
        <h1 class="slds-page-header__title slds-m-right_small
            slds-truncate slds-align-left">Create New Contact</h1>
    </div>

    <!-- Display the new contact form -->
     <lightning:input aura:id="contactField" name="firstName" label="First Name"
                      value="{!v.newContact.FirstName}" required="true"/>

    <lightning:input aura:id="contactField" name="lastname" label="Last Name"
```

```
                    value="{!v.newContact.LastName}" required="true"/>

    <lightning:input aura:id="contactField" name="title" label="Title"
                    value="{!v.newContact.Title}" />

    <lightning:input aura:id="contactField" type="phone" name="phone" label="Phone
Number"
                    pattern="^(1?(-?\d{3})-?)?(\d{3})(-?\d{4})$"
                    messageWhenPatternMismatch="The phone number must contain 7, 10,
 or 11 digits. Hyphens are optional."
                    value="{!v.newContact.Phone}" required="true"/>

    <lightning:input aura:id="contactField" type="email" name="email" label="Email"
                    value="{!v.newContact.Email}" />

    <lightning:button label="Cancel" onclick="{!c.handleCancel}"
class="slds-m-top_medium" />
    <lightning:button label="Save Contact" onclick="{!c.handleSaveContact}"
              variant="brand" class="slds-m-top_medium"/>

</aura:component>
```

The component defines the following attributes, which are used as member variables.

- *account*—holds the full account record, after it's loaded in the init handler
- *newContact*—an empty contact, used to capture the form field values

The rest of the component definition is a standard form that displays an error on the field if the required fields are empty or the phone field doesn't match the specified pattern.

The component's controller has all of the interesting code, in three action handlers.

`quickContactController.js`

```
({
    doInit : function(component, event, helper) {

        // Prepare the action to load account record
        var action = component.get("c.getAccount");
        action.setParams({"accountId": component.get("v.recordId")});

        // Configure response handler
        action.setCallback(this, function(response) {
            var state = response.getState();
            if(state === "SUCCESS") {
                component.set("v.account", response.getReturnValue());
            } else {
                console.log('Problem getting account, response state: ' + state);
            }
        });
        $A.enqueueAction(action);
    },

    handleSaveContact: function(component, event, helper) {
        if(helper.validateContactForm(component)) {
            // Prepare the action to create the new contact
            var saveContactAction = component.get("c.saveContactWithAccount");
            saveContactAction.setParams({
                "contact": component.get("v.newContact"),
                "accountId": component.get("v.recordId")
            });

            // Configure the response handler for the action
            saveContactAction.setCallback(this, function(response) {
                var state = response.getState();
                if(state === "SUCCESS") {

                    // Prepare a toast UI message
                    var resultsToast = $A.get("e.force:showToast");
                    resultsToast.setParams({
                        "title": "Contact Saved",
                        "message": "The new contact was created."
                    });

                    // Update the UI: close panel, show toast, refresh account page
                    $A.get("e.force:closeQuickAction").fire();
                    resultsToast.fire();
                    $A.get("e.force:refreshView").fire();
                }
                else if (state === "ERROR") {
                    console.log('Problem saving contact, response state: ' + state);
                }
                else {
                    console.log('Unknown problem, response state: ' + state);
                }
            });

            // Send the request to create the new contact
            $A.enqueueAction(saveContactAction);
        }

    },

    handleCancel: function(component, event, helper) {
        $A.get("e.force:closeQuickAction").fire();
    }
})
```

The first action handler, `doInit`, is an init handler. Its job is to use the record ID that's provided via the `force:hasRecordId` interface and load the full account record. Note that there's nothing to stop this component from being used in an action on another object, like a lead, opportunity, or custom object. In that case, `doInit` will fail to load a record, but the form will still display.

The `handleSaveContact` action handler validates the form by calling a helper function. If the form isn't valid, the field-level errors are displayed. If the form is valid, then the action handler:

- Prepares the server action to save the new contact.
- Defines a callback function, called the *response handler*, for when the server completes the action. The response handler is discussed in a moment.

- Enqueues the server action.

The server action's response handler does very little itself. If the server action was successful, the response handler:

- Closes the action panel by firing the `force:closeQuickAction` event.
- Displays a "toast" message that the contact was created by firing the `force:showToast` event.
- Updates the record page by firing the `force:refreshView` event, which tells the record page to update itself.

This last item displays the new record in the list of contacts, once that list updates itself in response to the refresh event.

The `handleCancel` action handler closes the action panel by firing the `force:closeQuickAction` event.

The component helper provided here is minimal, sufficient to illustrate its use. You'll likely have more work to do in any production quality form validation code.

`quickContactHelper.js`

```
({
    validateContactForm: function(component) {
        var validContact = true;

        // Show error messages if required fields are blank
        var allValid = component.find('contactField').reduce(function (validFields,
inputCmp) {
            inputCmp.showHelpMessageIfInvalid();
            return validFields && inputCmp.get('v.validity').valid;
        }, true);

        if (allValid) {
            // Verify we have an account to attach it to
            var account = component.get("v.account");
            if($A.util.isEmpty(account)) {
                validContact = false;
                console.log("Quick action context doesn't have a valid account.");
            }
        }

        return(validContact);
    }
})
```

Finally, the Apex class used as the server-side controller for this component is deliberately simple to the point of being obvious.

`QuickContactController.apxc`

```
public with sharing class QuickContactController {

    @AuraEnabled
    public static Account getAccount(Id accountId) {
        // Perform isAccessible() checks here
        return [SELECT Name, BillingCity, BillingState FROM Account WHERE Id =
:accountId];
    }

    @AuraEnabled
    public static Contact saveContactWithAccount(Contact contact, Id accountId) {
        // Perform isAccessible() and isUpdateable() checks here
        contact.AccountId = accountId;
```

```
            upsert contact;
            return contact;
        }

}
```

One method retrieves an account based on the record ID. The other associates a new contact record with an account, and then saves it to the database.

SEE ALSO:

Configure Components for Custom Actions

Create an Email as a Quick Action

In a custom component, create a button to launch the email composer with pre-populated content. To launch a record create a page with pre-populated field values, use the `lightning:pageReferenceUtils` and `lightning:navigation` components together.

These examples show you how to do this using standard actions and override actions.

Launch the QuickAction (Global) Send Email action from a custom component. Quick/Standard Actions can be called using page references and the navigation service API in any custom Aura component.

Define Navigation Services, pageReference Utils, and Action Button

Define the navigation services, the pageReference utils, and the action button in component markup.

```
<lightning:navigation aura:id="navService"/>
<lightning:pageReferenceUtils aura:id="pageRefUtil"/>
<div>
    <lightning:button label="Send an " value="Global.SendEmail" onclick="{!
c.openPageRefModal}"/>
</div>
```

Pass Attributes in `pageReference` to `navService`

Pass in the appropriate attributes in `pageReference` to `navService`.

```
openPageRefModal: function (cmp, event, helper) {

    var navService = cmp.find("navService");
    var actionApiName = event.getSource().get('v.value');
    var pageRef = {
        type: "standard__quickAction",
        attributes: {
            apiName : actionApiName
        },
        state: {
            recordId : '003xx000004WhEiAAK',
        }
    };
    navService.navigate(pageRef);
}
```

Add Predefined Fields Info

Allow the user to pass action field data as part of the `pageReference` attributes with the `fieldOverride` payload.

This code is an example of what a `pageReference` request could look like:

```
openPageRefModal: function (cmp, event, helper) {
    var navService = cmp.find("navService");
    var actionApiName = event.getSource().get('v.value');
    var pageRef = {
        type: "standard__quickAction",
        attributes: {
            apiName : actionApiName
        },
        state: {
            recordId : '003xx000004WhTJAA0'
        }
    };
    var defaultFieldValues = {
        HtmlBody: "Monthly Review",
        Subject : "Monthly Review"
    }
    pageRef.state.defaultFieldValues =
cmp.find("pageRefUtil").encodeDefaultFieldValues(defaultFieldValues);

    navService.navigate(pageRef);
}
```

Default Email Field Values

By default, Salesforce prepopulates the To field with a contact or lead email address when you open the email action from the contact or lead record home pages. Ensure that the fields you specify in the `encodeDefaultFieldValues` function aren't Read-Only in the Send Email global action's layout. If the HTML Body and Subject fields are Read-Only, the email draft doesn't include pre-populated text for those fields.

These fields are supported:

- `FromAddress`
- `ToAddress`
- `CCAddress`
- `BccAddress`
- `Subject`
- `HTMLBody`
- `RelatedToId`

The `FromAddress` field is set to the logged in user's email address. The `Subject` and `HTMLBody` are not set by default.

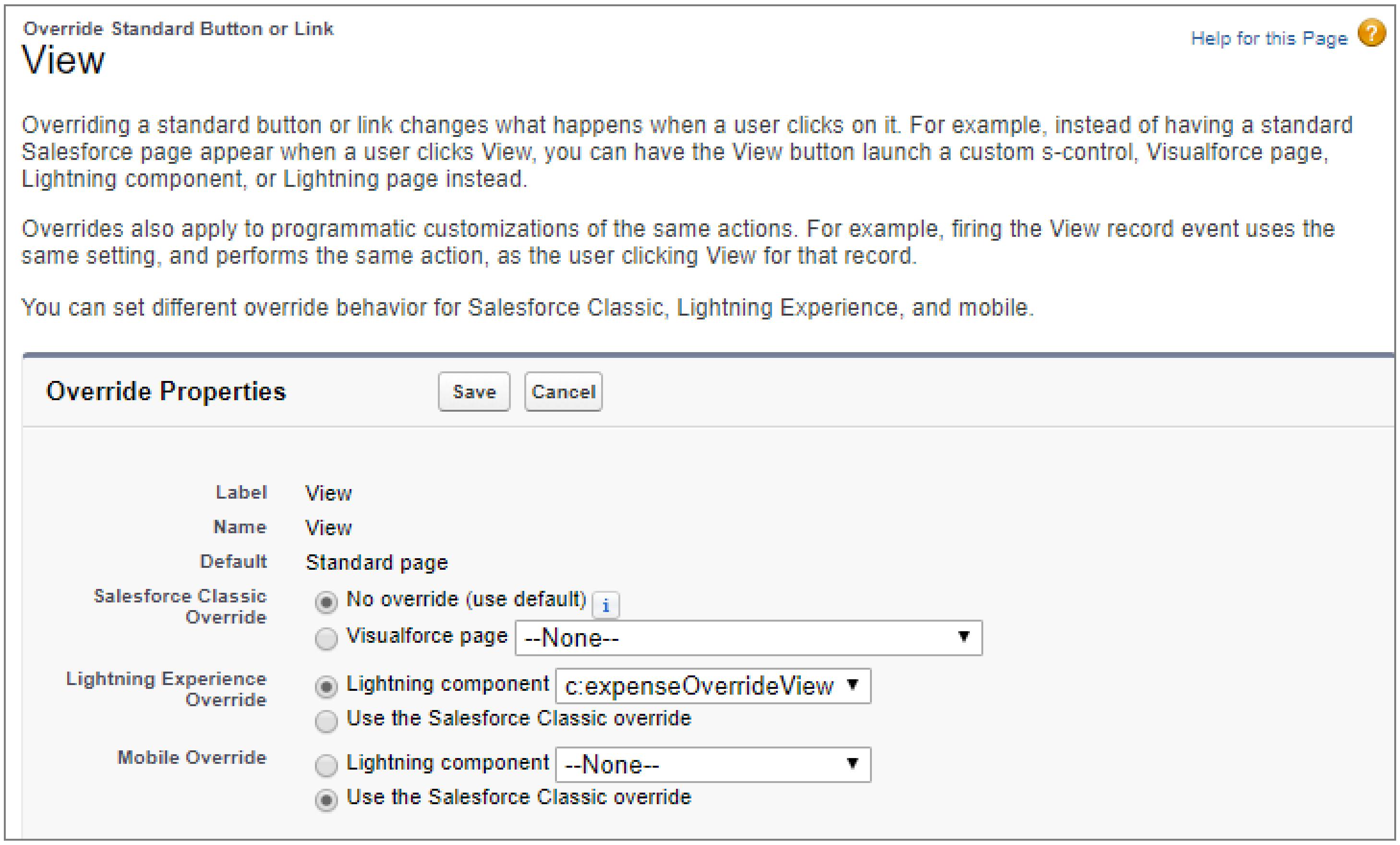
Only fields that are available on the email quick action are supported. For example, `AttachmentId` and `ContentDocumentIds` aren't supported as they are not part of the email quick action layout.

For more information on the supported fields, see Object Reference for the Salesforce Platform: EmailMessage.

Override Standard Actions with Aura Components

Add the `lightning:actionOverride` interface to an Aura component to enable the component to be used to override a standard action on an object. You can override the View, New, Edit, and Tab standard actions on most standard and all custom components. Overriding standard actions allows you to customize your org using Lightning components, including completely customizing the way you view, create, and edit records.

Overriding an action with an Aura component closely parallels overriding an action with a Visualforce page. Choose a Lightning component instead of a Visualforce page in the Override Properties for an action.



View

Overriding a standard button or link changes what happens when a user clicks on it. For example, instead of having a standard Salesforce page appear when a user clicks View, you can have the View button launch a custom s-control, Visualforce page, Lightning component, or Lightning page instead.

Overrides also apply to programmatic customizations of the same actions. For example, firing the View record event uses the same setting, and performs the same action, as the user clicking View for that record.

You can set different override behavior for Salesforce Classic, Lightning Experience, and mobile.

Override Properties

Label	View
Name	View
Default	Standard page
Salesforce Classic Override	[✓ No override (use default)] Visualforce page --None--
Lightning Experience Override	[✓ Lightning component c:expenseOverrideView] Use the Salesforce Classic override
Mobile Override	Lightning component --None-- [✓ Use the Salesforce Classic override]

However, there are important differences from Visualforce in how you create Lightning components that can be used as action overrides, and significant differences in how Salesforce uses them. You'll want to read the details thoroughly before you get started, and test carefully in your sandbox or Developer Edition org before deploying to production.

IN THIS SECTION:

Standard Actions and Overrides Basics

There are six standard actions available on most standard and all custom objects: Tab, List, View, Edit, New, and Delete. In Salesforce Classic, these are all distinct actions.

Override a Standard Action with an Aura Component

You can override a standard action with an Aura component in both Lightning Experience and mobile.

Creating an Aura Component for Use as an Action Override

Add the `lightning:actionOverride` interface to an Aura component to allow it to be used as an action override in Lightning Experience or the Salesforce mobile app. Only components that implement this interface appear in the **Lightning component** menu of an object action Override Properties panel.

Packaging Action Overrides

Action overrides for custom objects are automatically packaged with the custom object. Action overrides for standard objects can't be packaged.

Standard Actions and Overrides Basics

There are six standard actions available on most standard and all custom objects: Tab, List, View, Edit, New, and Delete. In Salesforce Classic, these are all distinct actions.

Lightning Experience and the Salesforce mobile app combine the Tab and List actions into one action, Object Home. However, Object Home is reached via the Tab action in Lightning Experience, and the List action in the Salesforce mobile app. Finally, the Salesforce mobile app has a unique Search action (reached via Tab). (Yes, it's a bit awkward and complicated.)

This table lists the standard actions you can override for an object as the actions are named in Setup, and the resulting action that's overridden in the three different user experiences.

Override in Setup	Salesforce Classic	Lightning Experience	Mobile
Tab	object tab	object home	search
List	object list	**n/a**	object home
View	record view	record home	record home
Edit	record edit	record edit	record edit
New	record create	record create	record create
Delete	record delete	record delete	record delete

Note:

- "n/a" doesn't mean you can't *access* the standard behavior, and it doesn't mean you can't *override* the standard behavior. It means you can't *access the override*. It's the override's functionality that's not available.
- There are two additional standard actions, Accept and Clone. These actions are more complex, and overriding them is an advanced project. Overriding them isn't supported.

How and Where You Can Use Action Overrides

Aura components can be used to override the View, New, New Event, Edit, and Tab standard actions in Lightning Experience and the Salesforce mobile app. You can use an Aura component as an override in Lightning Experience and mobile, but not Salesforce Classic.

Override a Standard Action with an Aura Component

You can override a standard action with an Aura component in both Lightning Experience and mobile.

You need at least one Aura component in your org that implements the `lightning:actionOverride` interface. You can use a custom component of your own, or a component from a managed package.

Go to the object management settings for the object with the action you plan to override.

1. Select **Buttons, Links, and Actions**.
2. Select **Edit** for the action you want to override.
3. Select **Lightning component** for the area you want to set the override.
4. From the drop-down menu, select the name of the Lightning component to use as the action override.
5. Select **Save**.

Note: Users won't see changes to action overrides until they reload Lightning Experience or the Salesforce mobile app.

SEE ALSO:

Salesforce Help: Find Object Management Settings

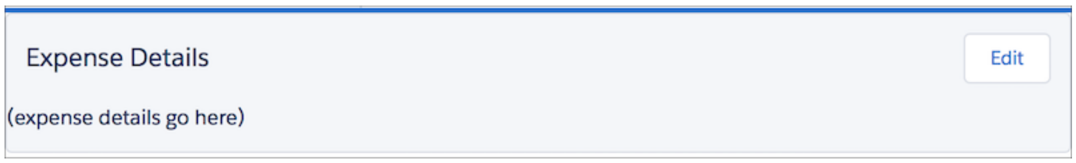
Salesforce Help: Override Standard Buttons and Tab Home Pages

Creating an Aura Component for Use as an Action Override

Add the `lightning:actionOverride` interface to an Aura component to allow it to be used as an action override in Lightning Experience or the Salesforce mobile app. Only components that implement this interface appear in the **Lightning component** menu of an object action Override Properties panel.

```
<aura:component
    implements="lightning:actionOverride,force:hasRecordId,force:hasSObjectName">

    <article class="slds-card">
      <div class="slds-card__header slds-grid">
        <header class="slds-media slds-media_center slds-has-flexi-truncate">
          <div class="slds-media__body">
            <h2><span class="slds-text-heading_small">Expense Details</span></h2>
          </div>
        </header>
        <div class="slds-no-flex">
            <lightning:button label="Edit" onclick="{!c.handleEdit}"/>
        </div>
      </div>
      <div class="slds-card__body">(expense details go here)</div>
    </article>
</aura:component>
```

In Lightning Experience, the standard Tab and View actions display as a page, while the standard New and Edit actions display in an overlaid panel. When used as action overrides, Aura components that implement the `lightning:actionOverride` interface replace the standard behavior completely. However, overridden actions always display as a page, not as a panel. Your component displays without controls, except for the main Lightning Experience navigation bar. Your component is expected to provide a complete user interface for the action, including navigation or actions beyond the navigation bar.


Expense Details

Edit

(expense details go here)

One important difference from Visualforce that's worth noting is how components are added to the **Lightning component** menu. The **Visualforce page** menu lists pages that either use the standard controller for the specific object, or that don't use a standard controller at all. This filtering means that the menu options vary from object to object, and offer only pages that are specific to the object, or completely generic.

The **Lightning component** menu includes every component that implements the `lightning:actionOverride` interface. A component that implements `lightning:actionOverride` can't restrict an admin to overriding only certain actions, or only for certain objects. We recommend that your organization adopt processes and component naming conventions to ensure that components are used to override only the intended actions on intended objects. Even so, it's your responsibility as the component developer to ensure that components that implement the `lightning:actionOverride` interface gracefully respond to being used with any action on any object.

Access Current Record Details

Components you plan to use as action overrides usually need details about the object type they're working with, and often the ID of the current record. Your component can implement the following interfaces to access those object and record details.

force:hasRecordId

Add the `force:hasRecordId` interface to an Aura component to enable the component to be assigned the ID of the current record. The current record ID is useful if the component is used on a Lightning record page, as an object-specific custom action or action override in Lightning Experience or the Salesforce mobile app, and so on.

force:hasSObjectName

Add the `force:hasSObjectName` interface to an Aura component to enable the component to be assigned the API name of current record's sObject type. The sObject name is useful if the component can be used with records of different sObject types, and needs to adapt to the specific type of the current record.

Note: As of Spring '19 (API version 45.0), you can build Lightning components using two programming models: the Lightning Web Components model, and the original Aura Components model. Lightning web components are custom HTML elements built using HTML and modern JavaScript. Lightning web components and Aura components can coexist and interoperate on a page. As of API version 45.0, when we say Lightning components, we mean both Aura components and Lightning web components.

Packaging Action Overrides

Action overrides for custom objects are automatically packaged with the custom object. Action overrides for standard objects can't be packaged.

When you package a custom object, overrides on that object's standard actions are packaged with it. This includes any Lightning components used by the overrides. Your experience should be "it just works."

However, standard objects can't be packaged. As a consequence, there's no way to package overrides on the object's standard actions.

To override standard actions on standard objects in a package, do the following:

- Manually package any Lightning components that are used by the overrides.
- Provide instructions for subscribing orgs to manually override the relevant standard actions on the affected standard objects.

SEE ALSO:

Override a Standard Action with an Aura Component

Metadata API Developer Guide : ActionOverride

Navigate Across Your Apps with Page References

The `pageReference` JavaScript object represents a URL for a page. You can use a `pageReference` instead of parsing or creating a URL directly. This approach helps you avoid broken navigation if Salesforce changes URL formats in the future.

These navigation resources are supported only in Lightning Experience, Experience Builder sites, and the Salesforce mobile app. They're not supported in other containers, such as Lightning Components for Visualforce, or Lightning Out. This is true even if you access these containers inside Lightning Experience or the Salesforce mobile app.

Basic Navigation

The `pageReference` JavaScript object represents a URL for a page. You can use a `pageReference` instead of parsing or creating a URL directly. This approach helps you avoid broken navigation if Salesforce changes URL formats in the future.

Use the following resources to simplify navigation across your apps. URLs for components using these resources are case-sensitive. For examples, see the Component Library.

Important: Navigation isn't supported for inactive pages. A page is considered inactive if it's not currently visible in the DOM, such as a minimized page.

`lightning:navigation`

To navigate to a page or component, use the `navigate(pageReference, replace)` method from `lightning:navigation`. This approach is a substitute for a `navigateTo*` event, and both are supported.

When you navigate to a page reference from a modal, such as from a component that's enabled for quick actions, the modal isn't automatically closed by default. To automatically close the modal when navigating to another page reference, set `replace` to `true`.

To generate a URL in your component, use the `generateUrl()` method in `lightning:navigation` to resolve the URL.

Note: `generateUrl()` returns a promise, which calls back with the resulting URL.

`lightning:isUrlAddressable`

To enable a component to navigate directly via a URL, add the `lightning:isUrlAddressable` interface to your component.

Tip: `pageReference` and `lightning:isUrlAddressable` replace the `force:navigateToComponent` event for navigating directly to a component. Unlike the `force:navigateToComponent` event information-mapping protocol, the only attribute populated through the navigation dispatching system is the `pageReference` attribute. Information is passed to the addressed component through the `state` properties on the target page reference. `lightning:isUrlAddressable` doesn't automatically set attributes on the target component. Get parameters from `v.pageReference.state` and manually set them using the target component's `init` handler.

`pageReference` provides a well-defined structure that describes the page type and its corresponding attributes. `pageReference` supports the following properties.

Property	Type	Description	Required?
`type`	String	The API name of the `pageReference` type, for example, `standard__objectPage`.	Y
`attributes`	Object	Values for each attribute specified by the page definition, for example, `objectAPIName` or `actionName`.	Y
`state`	Object	Parameters that are tied to the query string of the URL in Lightning Experience, such as `filterName`. The routing framework doesn't depend on `state` to render a page. Some page reference types support a standard set of `state` properties. You can also pass non-standard properties into `state` as long as they're namespaced. Note: Experience Builder sites don't support the `state` property.	

SEE ALSO:

pageReference Types

Add Links to Lightning Pages from Your Custom Components

To link to Lightning Experience pages, use `lightning:formattedUrl` in your custom component. The `lightning:formattedUrl` component displays a URL as a hyperlink.

If you use raw anchor tags or the `ui:outputUrl` (deprecated) component for links, the page does a full reload each time you click the link. To avoid full page reloads, replace your link components with `lightning:formattedUrl`.

For examples, see the Component Library.

Migrate from `ui:outputUrl` to `lightning:formattedUrl`

Copy the attributes from the `ui:outputUrl` component.

```
<aura:component>
    <ui:outputURL value="https://my/path" label="Contact ID" />
</aura:component>
```

Paste the same attributes into the `lightning:formattedUrl` component. `lightning:formattedUrl` supports more attributes, like `tooltip`.

```
<aura:component>
    <div aura:id="container">
        <p><lightning:formattedUrl value="https://my/path" label="Contact ID" tooltip="Go
 to Contact's recordId" /></p>
    </div>
</aura:component>
```

SEE ALSO:

Component Library: lightning:formattedUrl Reference

Add Query Parameters

To add query parameters to the URL, update the PageReference `state` property. The key-value pairs of the `state` property are serialized to URL query parameters. The query parameters describe the page and form a more specific URL that the user can save or bookmark.

Keep these behaviors in mind when working with the `state` property.

- You can't directly change the `pageReference` object. To update the `state`, create a new `pageReference` object, and copy the values using `Object.assign({}, `***`pageReference`***`)`.
- `state` parameters must be namespaced. For example, a managed package with the namespace `abc` with a parameter `accountId` is represented as `abc__accountId`. The namespace for custom components is `c__`.Parameters without a namespace are reserved for Salesforce use. This namespace restriction is introduced under a critical update in Winter '19 and enforced in Summer '19.
- Since the key-value pairs of `PageReference.state` are serialized to URL query parameters, all the values must be strings.
- Code that consumes values from `state` must parse the value into its proper format.
- To delete a value from the `state` object, define it as `undefined`.

If your component uses the `lightning:hasPageReference` or `lightning:isUrlAddressable` interfaces, always implement a change handler. When the target of a navigation action maps to the same component, the routing container might simply update the `pageReference` attribute value instead of recreating the component. In this scenario, a change handler ensures that your component reacts correctly.

Navigate to a Record Create Page with Default Field Values

The `lightning:pageReferenceUtils` component provides utilities for encoding default field values into a string. Pass this string into the `pageReference.state.defaultFieldValues` attribute on `standard__objectPage` page reference types.

To launch a record create page with prepopulated field values, use the `lightning:pageReferenceUtils` and `lightning:navigation` components together. The examples on this page show you how to do this using standard actions and override actions.

Launch an Account Record with Default Field Values Using a Standard Action

This example adds two standard action links that navigate to a record create page with default field values. The first link uses a URL that you generate using `generateUrl(pageRef)`, and the second link navigates directly to the record create page using `navigate(pageRef)`.

```
<!-- auraNavigator.cmp -->
<aura:component implements="force:appHostable,flexipage:availableForAllPageTypes">
    <aura:attribute name="url" type="String"/>

    <!-- Specify the pageReference type. Only object is supported. -->
    <aura:attribute name="pageReference" type="Object"/>
    <aura:handler name="init" value="{! this }" action="{! c.init }"/>

    <!-- Implement the navigation service. -->
    <lightning:navigation aura:id="navService"/>

    <!-- pageReferenceUtil component -->
    <lightning:pageReferenceUtils aura:id="pageRefUtils"/>

    <!-- Generate a link to launch an account record create page. -->
    <a href="{!v.url}">New Account (Aura Link)</a> <br/>

    <!-- Launch an account record create page -->
    <a href="#" onclick="{!c.handleClick}">New Account (Aura PageRef)</a> <br/>
</aura:component>
```

In your client-side controller, get `defaultFieldValues` from `pageRef` and pass them into `encodeDefaultFieldValues()`. When you click a link to create an account, `encodeDefaultFieldValues()` reads and encodes the values and passes them into a new `standard__objectPage`.

```
// auraNavigatorController.js
({
    init : function(cmp, event, helper) {
        var navService = cmp.find("navService");
        var pageRef = {
            type: "standard__objectPage",
            attributes: {
                objectApiName: "Account",
                actionName: "new"
            },
            state: {
            }
        }
        // Replace with your own field values
        var defaultFieldValues = {
            Name: "Salesforce, #1=CRM",
            OwnerId: "005XXXXXXXXXXXXXXX",
            AccountNumber: "ACXXXX",
            NumberOfEmployees: 35000,
            CustomCheckbox__c: true
        };
        pageRef.state.defaultFieldValues =
cmp.find("pageRefUtils").encodeDefaultFieldValues(defaultFieldValues);
        cmp.set("v.pageReference", pageRef);
        var defaultUrl = "#";

        // Generate a Link for the Aura Link example
        navService.generateUrl(pageRef)
        .then($A.getCallback(function(url) {
            cmp.set("v.url", url ? url : defaultUrl);
        }), $A.getCallback(function(error) {
            cmp.set("v.url", defaultUrl);
        }));
    },

    // Navigate to the record create page for the Aura PageRef example
    handleClick : function(cmp, event, helper) {
        var navService = cmp.find("navService");
        var pageRef = cmp.get("v.pageReference");
        event.preventDefault();
        navService.navigate(pageRef);
    }
})
```

Handle Default Field Values Using an Override Action

With standard actions, the default field values pass through the URL to the object as a string, and the redirect and replace is handled for you. With override actions, the encoding is handled for you, but you are responsible for decoding the string of default field values from the URL and handling the redirect and replace.

Important: We recommend that you always redirect and replace to remove the default field values from the URL and browser history.

This example uses `hasPageReference` to launch an account create page via an override action.

```
<!-- auraNewAccountOverride.cmp -->
<aura:component implements="lightning:actionOverride,lightning:hasPageReference">

    <lightning:pageReferenceUtils aura:id="pageRefUtils"/>
    <lightning:recordEditForm
                    objectApiName="Account"
                    onload="{!c.handleCreateLoad}">
        <lightning:messages />
        <lightning:inputField aura:id="nameField" fieldName="Name"/>
        <lightning:inputField aura:id="numOfEmpField" fieldName="NumberOfEmployees"/>
        <lightning:inputField aura:id="ownerIdField" fieldName="OwnerId"/>
        <lightning:inputField aura:id="customCheckField" fieldName="CustomCheckbox__c"/>
        <lightning:button class="slds-m-top_small" type="submit" label="Create new" />
    </lightning:recordEditForm>
</aura:component>
```

The client-side controller reads the default field values from the `state` key and gets the encoded string. It then passes the string into `decodeDefaultFieldValues()` to decode it and retrieve the object.

Important: All default field values are passed into the record create page as strings, regardless of field type. For example, `NumberOfEmployees: 35000` is passed into the page as the string `35000` instead of a number field type. Boolean values are passed into the page as `true` or `false` strings.

This example is similar to prepopulating field values using `lightning:recordEditForm`, except that here the `defaultFieldValues` are dynamically generated when navigating to the form.

```
// auraNewAccountOverrideController.js
({
    handleCreateLoad: function (cmp, event, helper) {
        var pageRef = cmp.get("v.pageReference");
        var defaultFieldValues =
       cmp.find("pageRefUtils").decodeDefaultFieldValues(pageRef.state.defaultFieldValues);


       var nameFieldValue = cmp.find("nameField").set("v.value", defaultFieldValues.Name);

        var numOfEmpFieldValue = cmp.find("numOfEmpField").set("v.value",
defaultFieldValues.NumberOfEmployees);
        var ownerIdFieldValue = cmp.find("ownerIdField").set("v.value",
defaultFieldValues.OwnerId);
        var customCheckFieldValue = cmp.find("customCheckField").set("v.value",
defaultFieldValues.CustomCheckbox__c === 'true');
    }
})
```

SEE ALSO:

Component Library: lightning:pageReferenceUtils Reference

Navigate to a Web Page

The navigation service supports different kinds of pages in Lightning. Each page reference type supports a different set of attributes and state properties.

Instead of using `force:navigateToURL`, we recommend navigating to web pages using the `lightning:navigate` component with the `standard__webPage` page type.

This code shows examples of navigating to a web page using the old `force:navigateToURL` event.

```
// Old way to navigate to a web page
$A.get("markup://force:navigateToURL").setParams({
    url: 'https://salesforce.com',
}).fire();
```

Replace the previous code that uses `force:navigateToURL` with the following code. This example shows how to navigate to a web page using the `standard__webPage` page type. It assumes that you added `<lightning:navigation aura:id="navigationService" />` in your component markup.

```
cmp.find("navigationService").navigate({
    type: "standard__webPage",
    attributes: {
        url: 'https://salesforce.com'
    }
});
```

Migrate to `lightning:isUrlAddressable` from `force:navigateToComponent`

The `pageReference` JavaScript object represents a URL for a page. You can use a `pageReference` instead of parsing or creating a URL directly. This approach helps you avoid broken navigation if Salesforce changes URL formats in the future.

If you're currently using the `force:navigateToComponent` event, you can provide backward compatibility for bookmarked links by redirecting requests to a component that uses `lightning:isUrlAddressable`.

First, copy your original component, including its definition, controller, helper, renderer, and CSS. Make the new component implement the `lightning:isUrlAddressable` interface.

Change the new component to read the values passed through the navigation request from `cmp.get("v.pageReference").state`.

Note: You can't use two-way binding to map values from `pageReference.state` to a subcomponent that sets those values. You can't modify the `state` object. As a workaround, copy the values from `pageReference.state` into your own component's attribute using a handler.

```
// Add a handler to your component
<aura:handler name="init" value="{!this}" action="{!c.init}" />

// Controller example
({
    init: function(cmp, event, helper) {
        var pageReference = cmp.get("v.pageReference");
        cmp.set("v.myAttr", pageReference.state.c__myAttr);
        // myAttr can be modified, but isn't reflected in the URL
    }
})
```

In the new component, remove the attributes mapped from the URL that aren't used to copy values from the page state in the component's `init` handler.

Change the instances that navigate to your old component to the new API and address of your new component. For example, remove instances of `force:navigateToComponent`, like `$A.get("e.force:navigateToComponent").setParams({componentDef: "c:oldCmp", attributes: {"myAttr": "foo"}}).fire();`.

Add `<lightning:navigation aura:id="navigationService" />` to your component markup, and update it to use `navigationService`. Pass in a `pageReference`.

```
cmp.find("navigationService").navigate({
    type: "standard__component",
    attributes: {
        componentName: "c__myCmpCopy"
    },
    state: {
        "c__myAttr": "foo"
    }
});
```

In the original component's `init` handler, send a navigation redirect request to navigate to the new component. Pass the third argument in the navigate API call as `true`. This argument indicates that the request replaces the current entry in the browser history and avoids an extra entry when using a browser's navigation buttons.

```
({
    init: function(cmp, event, helper) {
        cmp.find("navigation").navigate({
            type: "standard__component",
            attributes: {
                componentName: "c__componentB" },
            state: {
                c__myAttr: cmp.get("v.myAttr")
            }
        }, true); // replace = true
    }
})
```

Remove all other code from the original component's definition, controller, helper, renderer, and CSS. Leave only the navigation redirect call.

`pageReference` Types

To navigate in Lightning Experience, Experience Builder sites, or the Salesforce mobile app, define a `PageReference` object. The `pageReference` type generates a unique URL format and defines attributes that apply to all pages of that type. For Experience Builder sites, depending on the page type, the `pageReference` property requirements can differ between LWR sites and Aura sites.

The following types are supported.

- App
- External Record Page
- External Record Relationship Page
- Knowledge Article
- Lightning Component (must implement `lightning:isUrlAddressable`)
- Login Page
- Managed Content Page (Salesforce CMS)
- Named Page (Experience Cloud)
- Named Page (Standard)
- Navigation Item Page
- Object Page
- Record Page
- Record Relationship Page
- Web Page

Note: `PageReference` objects are supported on a limited basis for Experience Builder sites, as noted for each type.

App Type

A standard or custom app available from the App Launcher in an org. Use this `pageReference` type to create custom navigation components that take users to a specific app or page within the app. Connected apps aren't supported.

Note: If you're navigating users to a different app using a `pageRef`, the app opens in the same window by default. To open a link in a new tab, see the navigation service documentation.

Type

```
standard__app
```

Experience

Lightning Experience

Type Attributes

Property	Type	Description	Required?
`appTarget`	String	App that you're navigating to. Pass either the `appId` or `appDeveloperName` to the `appTarget`. The `appId` is the `DurableId` field on the `AppDefinition` sObject. The `appDeveloperName` value is formed by concatenating the app's namespace with the developer name. To find the app's developer name, navigate to the App Manager in Setup and look in the Developer Name column. For standard apps, the namespace is *`standard__`*. For custom apps, it's *`c__`*. For managed packages, it's the namespace registered for the package.	Yes
pageRef	PageReference	Identifies a specific location in the app you're navigating to. Pass in the `pageRef` and applicable attributes for that `pageRef` type.	No

Example Navigating to an App

```
{
    type: "standard__app",
    attributes: {
        appTarget: "standard__Sales",
    }
}
```

Example Navigating to a Record in an App

```
{
    type: "standard__app",
    attributes: {
        appTarget: "standard__LightningSales",
        pageRef: {
            type: "standard__recordPage",
            attributes: {
                recordId: "001xx000003DGg0AAG",
                objectApiName: "Account",
                actionName: "view"
            }
        }
    }
}
```

URL Format

```
/lightning/app/{appTarget}{...pageRef}
```

URL Format Examples

Navigate to the app's homepage using the `appId`

```
/lightning/app/06mRM0000008dNrYAI
```

Navigate to an object record's page in the app using the `appId`

```
/lightning/app/06mRM0000008dNrYAI/o/Case/home
```

Navigate to the app's homepage using the `appDeveloperName`

```
/lightning/app/standard__LightningSales
```

Navigate to an object record's page in the app using the `appDeveloperName`

```
/lightning/app/standard__LightningSales/o/Case/home
```

External Record Page

A page that interacts with an external record. Currently supports CMS Connect pages.

Type

```
comm__externalRecordPage
```

Experience

Experience Builder Aura Sites

Type Attributes

Property	Type	Description	Required?
`recordId`	String	External record ID.	
`objectType`	String	External record type. Currently only supports *`cms`* for CMS Connect.	
`objectInfo`	Object	Additional information used to identify the record for the `objectType`.	

Example

```
{
    type: "comm__externalRecordPage",
    attributes: {
        recordId: "26",
        objectType: "cms",
        objectInfo: {
            cmsSourceName: "blog",
            cmsTypeName: "feed",
        }
    },
    state: {
        recordName: "coffee-on-the-world-map",
    }
}
```

URL Format

```
/{baseUrl}/{recordId}/{recordName}
```

External Record Relationship Page

A page that interacts with an external relationship on a particular record in the org. Currently only supports Quip Related List page.

Type

```
comm__externalRecordRelationshipPage
```

Experience

Experience Builder Aura Sites

Type Attributes

Property	Type	Description	Required?
`recordId`	String	The 18 character record ID.	
`objectType`	String	External record type. Currently only supports *`quip`* for Quip docs.	

Example

```
{
    type: "comm__externalRecordRelationshipPage",
    attributes: {
        recordId: "001xx000003DGg0AAG",
        objectType: "quip",

```

URL Format

```
/{baseUrl}/{recordId}
```

Lightning Component Type

A Lightning component that implements the `lightning:isUrlAddressable` interface, which enables the component to be navigated directly via URL.

Type

```
standard__component
```

Experience

Lightning Experience, Salesforce Mobile App

Type Attributes

Property	Type	Description	Required?
`componentName`	String	The Lightning component name in the format `namespace__componentName`.	Yes

Example

```
{
    "type": "standard__component",
    "attributes": {
        "componentName": "c__MyLightningComponent"
    },
    "state": {
        "c__myAttr": "attrValue"
    }
}
```

You can pass any key and value in the `state` object. The key must include a namespace, and the value must be a string. If you don't have a registered namespace, add the default namespace of `c__`.

URL Format

```
/cmp/{componentName}?c__myAttr=attrValue
```

Login Page Type

An authentication for an Experience Builder site.

Type

```
comm__loginPage
```

Experience

Experience Builder sites

Type Attributes

Property	Type	Description	Required
`actionName`	String	A login-related action to be performed. Possible values are: • `login` • `logout`	Yes

Note: You can only navigate to the following

```
comm__namedPages
```

when you're calling navigate from them: Login, Check Password, Forgot Password, Login Error, and Register. Other page references don't work from these pages.

Example

```
{
    type: "comm__loginPage",
    attributes: {
        actionName: "login"
    }
}
```

Knowledge Article Page Type

A page that interacts with a Knowledge Article record.

Type

```
standard__knowledgeArticlePage
```

Experience

Lightning Experience, Experience Builder sites, Salesforce Mobile App

Type Attributes

Property	Type	Description	Required?
`articleType`	String	The `ArticleType` API name of the Knowledge Article record. In Experience Builder sites, `articleType` is ignored.	Yes
`urlName`	String	The value of the `urlName` field on the target KnowledgeArticleVersion record. The `urlName` is the article's URL.	Yes

Example

```
{
    "type": "standard__knowledgeArticlePage",
    "attributes": {
        "articleType": "Briefings",
        "urlName": "February-2017"
    }
}
```

URL Format

```
/articles/{articleType}/{urlName}
```

URL Format (Experience Cloud)

```
/article/{urlName}
```

Managed Content Page (Salesforce CMS)

A CMS content page in an Experience Builder site with a unique name.

Type

```
standard__managedContentPage
```

Experience

Experience Builder sites

Type Attributes

Property	Type	Description	Required?
contentTypeName	String	The name of the Salesforce CMS content type.	Yes
contentKey		The unique content key that identifies CMS content.	Yes

Example

```
{
    type: 'standard__managedContentPage',
    attributes : {
        'contentTypeName': 'news',
        'contentKey': 'MCOMALJDRAYFFSFPNBQONYXVFHOA'
    }
}
```

URL Format

```
/:urlAlias
```

Named Page Type (Experience Cloud)

A standard page in an Experience Builder site with a unique name. If an error occurs, the error view loads and the URL isn't updated.

Type

```
comm__namedPage
```

Experience

Experience Builder sites

Type Attributes

Property	Type	Description	Required?
name	String	The unique name of the Experience Builder site page. The value for `name` is the API Name value for a supported page. The **API Name** field can only be defined when a new page is being created, and must be unique. If the **API Name** isn't defined upon page creation, it's automatically generated. The value of `home` is reserved for the landing page of any Experience Builder site in your org. Supported pages are: • Home • Account Management	Yes

Property	Type	Description	Required?
		• Contact Support • Error • Login • My Account • Top Articles • Topic Catalog • Custom pages	

Example

```
{
    type: "comm__namedPage",
    attributes: {
        name: "Home"
    }
}
```

URL Format

```
/{URL as defined on the page’s properties}
```

Named Page Type (Standard)

A standard page with a unique name. If an error occurs, the error view loads and the URL isn’t updated.

Type

```
standard__namedPage
```

Experience

Lightning Experience, Salesforce Mobile App

Type Attributes

Property	Type	Description	Required?
`pageName`	String	The unique name of the page. Possible values are: • `home` • `chatter` • `today` • `dataAssessment` • `filePreview`	Yes

Example

```
{
    "type": "standard__namedPage",
    "attributes": {
        "pageName": "home"
    }
}
```

URL Format

```
/page/{pageName}
```

Navigation Item Page Type

A page that displays the content mapped to a CustomTab. Visualforce tabs, web tabs, Lightning Pages, and Lightning Component tabs are supported.

Type

```
standard__navItemPage
```

Experience

Lightning Experience, Salesforce Mobile App

Type Attributes

Property	Type	Description	Required?
`apiName`	String	The unique name of the CustomTab.	Yes

Example

```
{
    "type": "standard__navItemPage",
    "attributes": {
        "apiName": "MyCustomTabName"
    }
}
```

URL Format

```
/n/{apiName}
```

Object Page Type

A page that interacts with a standard or custom object in the org and supports standard actions for that object.

Note: The `standard__objectPage` type replaces the `force:navigateToObjectHome` and the `force:navigateToList` events.

Type

```
standard__objectPage
```

Experience

Lightning Experience, Experience Builder sites, Salesforce Mobile App

Type Attributes

Property	Type	Description	Required?
`actionName`	String	The action name to invoke. Valid values include `home`, `list`, and `new`. In Experience Builder sites, `list` and `home` are the same.	Yes
`objectApiName`	String	The API name of the standard or custom object. For custom objects that are part of a managed package, prefix the custom object with `ns__`.	Yes

State

Property	Type	Description	Supported Actions	Required?
`filterName`	String	ID or developer name of the object page. Default is `Recent`.	`list`	No
`defaultFieldValues`	String	List of key-value pairs for the default field values that you're passing. This list is generated by the `lightning:pageReferenceUtils` component. See lightning:pageReferenceUtils for details.	`new`	No
`nooverride`	String	To use a standard action, assign this property any value, such as `1`. To use an override action, don't include this property at all.	`home`, `list`, `new`	No

Standard Object Example

```
// Opens the case home page.
{
    "type": "standard__objectPage",
    "attributes": {
        "objectApiName": "Case",
        "actionName": "home"
    }
}
```

Navigate to a Specific List View Example

```
// Navigates to account list with the filter set to RecentlyViewedAccounts.
{
    "type": "standard__objectPage",
    "attributes": {
        "objectApiName": "ns__Widget__c",
        "actionName": "list"
    },
    "state": {
        "filterName": "RecentlyViewedAccounts"
  }
}
```

Navigate to a Record Create Page with Default Field Values

```
// Navigates to a new account object using these default field values.
//{
//    Name: "Salesforce, #1=CRM",
//    OwnerId: "005XXXXXXXXXXXXXXX",
//    AccountNumber: "ACXXXX",
//    NumberOfEmployees: 35000,
//    CustomCheckbox__c: true
//}
{
    type: "standard__objectPage",
    attributes: {
        objectApiName: "Account",
        actionName: "new"
    },
    state: {
        defaultFieldValues =
"AccountNumber=ACXXXX,CustomCheckbox__c=true,Name=Salesforce%2C%20%231%3DCRM,NumberOfEmployees=35000,OwnerId=005XXXXXXXXXXXXXXX",

        nooverride: "1"
    }
}
```

URL Format

```
/o/{objectApiName}/{actionName}
/o/{objectApiName}/{actionName}?filterName=Recent
```

URL Format (Experience Cloud)

```
/recordlist/{objectApiName}
/{baseUrl}/{objectApiName}
```

Record Page Type

A page that interacts with a record in the org and supports standard actions for that record.

Note: The `standard__recordPage` type replaces the `force:navigateToSObject` event.

Type

```
standard__recordPage
```

Experience

Lightning Experience, Experience Builder sites, Salesforce Mobile App

Type Attributes

Property	Type	Description	Required?
`actionName`	String	The action name to invoke. Valid values include `clone`, `edit`, and `view`. Experience Builder sites don't support the values `clone` or `edit`.	Yes

Property	Type	Description	Required?
objectApiName	String	The API name of the record's object. Optional for lookups.	Yes, for Experience Builder LWR sites only No, for all other experiences
recordId	String	The 18 character record ID.	Yes

State

Property	Type	Description	Required?
nooverride	String	To use a standard action, assign this property any value, such as `1`. To use an override action, don't include this property at all.	No

Example

```
{
        "type": "standard__recordPage",
        "attributes": {
            "recordId": "001xx000003DGg0AAG",
            "objectApiName": "PersonAccount",
            "actionName": "view"
        }
}
```

URL Format

```
/r/{objectApiName}/{recordId}/{actionName}
/r/{recordId}/{actionName}
```

URL Format (Experience Cloud)

```
/detail/{recordId}
/{baseUrl}/{recordId}
```

Record Relationship Page Type

A page that interacts with a relationship on a particular record in the org. Only related lists are supported.

Note: The `standard__recordRelationshipPage` type replaces the `force:navigateToRelatedList` event.

Type

```
standard__recordRelationshipPage
```

Experience

Lightning Experience, Experience Builder sites, Salesforce Mobile App

Type Attributes

Property	Type	Description	Required?
`actionName`	String	The action name to invoke. Only `view` is supported.	Yes
`objectApiName`	String	The API name of the object that defines the relationship. Optional for lookups.	Yes, for Experience Builder LWR sites only No, for all other experiences
`recordId`	String	The 18 character record ID of the record that defines the relationship.	Yes
`relationshipApiName`	String	The API name of the object's relationship field.	Yes

Example

```
{
    "type": "standard__recordRelationshipPage",
    "attributes": {
        "recordId": "500xx000000Ykt4AAC",
        "objectApiName": "Case",
        "relationshipApiName": "CaseComments",
        "actionName": "view"
    }
}
```

URL Format

```
/r/{objectApiName}/{recordId}/related/{relationshipApiName}/{actionName}
/r/{recordId}/related/{relationshipApiName}/{actionName}
```

URL Format (Experience Cloud)

```
/relatedlist/{recordId}/{relationshipApiName}
/{baseUrl}/related/{recordId}/{relationshipApiName}
```

Web Page

An external URL. Navigate to web pages using the `lightning:navigate` component with the `standard__webPage` page type instead of using `force:navigateToURL`. In Aura sites, certain internal Salesforce URLs have site-specific processing. For example, `/apex/` URLs are translated to `/sfdcpage/`. The Visualforce page is embedded within the site in an iFrame, which is the same behavior as with `force:navigateToURL`. Use `window.open` if you want to go straight to the URL, such as opening `/apex/` directly in a new tab.

Type

```
standard__webPage
```

Experience

Lightning Experience, Salesforce Mobile App

Attributes

Property	Type	Description	Required
`url`	String	The URL of the page you're navigating to.	Yes

Example

```
{
    "type": "standard__webPage",
    "attributes": {
        "url": "https://salesforce.com"
    }
}
```

URL Format

A web page opens as is in a new tab, so it doesn't have a URL format.

SEE ALSO:

LWR Sites for Experience Cloud: Lightning Navigation

Get Your Aura Components Ready to Use on Lightning Pages

Custom Aura components don't work on Lightning pages or in the Lightning App Builder right out of the box. To use a custom component in either of these places, configure the component and its component bundle so that they're compatible.

IN THIS SECTION:

Configure Components for Lightning Pages and the Lightning App Builder

There are a few steps to take before you can use your custom Aura components in either Lightning pages or the Lightning App Builder.

Configure Components for Lightning Experience Record Pages

After your component is set up to work on Lightning pages and in the Lightning App Builder, use these guidelines to configure the component so it works on record pages in Lightning Experience.

Create Components for the Outlook and Gmail Integrations

Create custom Aura components that are available to add to the email application pane for the Outlook and Gmail integrations.

Create Components for Forecast Pages

Create custom Aura components that are available to add to Lightning forecasts pages.

Create Dynamic Picklists for Your Custom Components

You can expose a component property as a picklist when the component is configured in the Lightning App Builder. The picklist's values are provided by an Apex class that you create.

Create a Custom Lightning Page Template Component

Every standard Lightning page is associated with a default template component, which defines the page's regions and what components the page includes. Custom Lightning page template components let you create page templates to fit your business needs with the structure and components that you define. Once implemented, your custom template is available in the Lightning App Builder's new page wizard for your page creators to use.

Lightning Page Template Component Best Practices

Keep these best practices and limitations in mind when creating Lightning page template components.

Make Your Lightning Page Components Width-Aware with lightning:flexipageRegionInfo

When you add a component to a region on a page in the Lightning App Builder, the `lightning:flexipageRegionInfo` sub-component passes the width of that region to its parent component. With `lightning:flexipageRegionInfo` and some strategic CSS, you can tell the parent component to render in different ways in different regions at runtime.

Tips and Considerations for Configuring Components for Lightning Pages and the Lightning App Builder

Keep these guidelines in mind when creating components and component bundles for Lightning pages and the Lightning App Builder.

Configure Components for Lightning Pages and the Lightning App Builder

There are a few steps to take before you can use your custom Aura components in either Lightning pages or the Lightning App Builder.

1. Add a New Interface to Your Component

To appear in the Lightning App Builder or on a Lightning page, a component must implement one of these interfaces.

Interface	Description
`flexipage:availableForAllPageTypes`	Makes your component available for record pages and any other type of page, including a Lightning app's utility bar.
`flexipage:availableForRecordHome`	If your component is designed for record pages only, implement this interface instead of `flexipage:availableForAllPageTypes`.
`clients:availableForMailAppAppPage`	Enables your component to appear on a Mail App Lightning page in the Lightning App Builder and in the Outlook and Gmail integrations.
`lightning:availableForForecastingPage`	Makes your component available for the forecasts page in Lightning.

Here's the sample code for a simple "Hello World" component.

```
<aura:component implements="flexipage:availableForAllPageTypes" access="global">
    <aura:attribute name="greeting" type="String" default="Hello" access="global" />
    <aura:attribute name="subject" type="String" default="World" access="global" />

    <div style="box">
      <span class="greeting">{!v.greeting}</span>, {!v.subject}!
    </div>
</aura:component>
```

Note: Mark your resources, such as a component, with `access="global"` to make the resource usable outside of your own org. For example, if you want a component to be usable in an installed package or by a Lightning App Builder user or an Experience Builder user in another org.

2. Add a Design Resource to Your Component Bundle

Use a design resource to control which attributes are exposed to builder tools like the Lightning App Builder, Experience Builder, or Flow Builder. A design resource lives in the same folder as your `.cmp` resource, and describes the design-time behavior of the Aura component—information that visual tools need to display the component in a page or app.

For example, if you want to restrict a component to one or more objects, set a default value on an attribute, or make an Aura component attribute available for administrators to edit in the Lightning App Builder, you need a design resource in your component bundle.

Here's the design resource that goes in the bundle with the "Hello World" component.

```
<design:component label="Hello World">
    <design:attribute name="subject" label="Subject" description="Name of the person you
want to greet" />
    <design:attribute name="greeting" label="Greeting" />
</design:component>
```

Design resources must be named *`componentName`*`.design`.

Optional: Add an SVG Resource to Your Component Bundle

You can use an SVG resource to define a custom icon for your component when it appears in the Lightning App Builder's component pane. Include it in the component bundle.

Here's a simple red circle SVG resource to go with the "Hello World" component.

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     width="400" height="400">
  <circle cx="100" cy="100" r="50" stroke="black"
    stroke-width="5" fill="red" />
</svg>
```

SVG resources must be named *`componentName`*`.svg`.

SEE ALSO:

Aura Component Bundle Design Resources

Tips and Considerations for Configuring Components for Lightning Pages and the Lightning App Builder

Component Bundles

Configure Components for Lightning Experience Record Pages

After your component is set up to work on Lightning pages and in the Lightning App Builder, use these guidelines to configure the component so it works on record pages in Lightning Experience.

Record pages are different from app pages in a key way: they have the context of a record. To make your components display content that is based on the current record, use a combination of an interface and an attribute.

- If your component is available for both record pages and any other type of page, implement `flexipage:availableForAllPageTypes`.

- If your component is designed only for record pages, implement the `flexipage:availableForRecordHome` interface instead of `flexipage:availableForAllPageTypes`.
- If your component needs the record ID, also implement the `force:hasRecordId` interface.

  Note: Don't expose the `recordId` attribute to the Lightning App Builder—don't put it in the component's design resource. You don't want admins supplying a record ID.

- If your component needs the object's API name, also implement the `force:hasSObjectName` interface.
- When a component is generated on the server from metadata, the page loads from a client-side cache of the indexed database. The client-side cache timeout is 8 hours, with a refresh interval of 15 minutes. This long timeout allows for faster page loads for users who bootstrap the application frequently or click links from outside the application to open a new browser window or tab to Lightning Experience.

Note: If your managed component implements the `flexipage` or `forceCommunity` interfaces, its upload is blocked if the component and its attributes aren't set to `access="global"`. For more information on access checks, see Controlling Access.

Note: When you use the Lightning App Builder, there is a known limitation when you edit a group page. Your changes appear when you visit the group from the Groups tab. Your changes don't appear when you visit the group from the Recent Groups list on the Chatter tab.

SEE ALSO:

Configure Components for Lightning Pages and the Lightning App Builder

Tips and Considerations for Configuring Components for Lightning Pages and the Lightning App Builder

Using Apex to Work with Salesforce Records

Create Components for the Outlook and Gmail Integrations

Create custom Aura components that are available to add to the email application pane for the Outlook and Gmail integrations.

To add a component to email application panes in the Outlook or Gmail integration, implement the `clients:availableForMailAppAppPage` interface.

To allow the component access to email or calendar events, implement the `clients:hasItemContext` interface.

The `clients:hasItemContext` interface adds attributes to your component that it can use to implement record- or context-specific logic. The attributes included are:

- The `source` attribute, which indicates the email or appointment source. Possible values include `email` and `event`.

```
<aura:attribute name="source" type="String" />
```

- The `mode` attribute, which indicates viewing or composing an email or event. Possible values include `view` and `edit`.

```
<aura:attribute name="mode" type="String" />
```

- The `people` attribute indicates recipients' email addresses on the current email or appointment.

```
<aura:attribute name="people" type="Object" />
```

  The shape of the `people` attribute changes according to the value of the `source` attribute.

When the source attribute is set to email, the people object contains the following elements.

```
{
    to: [ { name: nameString, email: emailString }, ... ],
    cc: [ ... ],
    from: { name: senderName, email: senderEmail },
}
```

When the source attribute is set to event, the people object contains the following elements.

```
{
    requiredAttendees: [ { name: attendeenameString, email: emailString }, ... ],
    optionalAttendees: [ { name: optattendeenameString, email: emailString }, ... ],
    organizer: { name: organizerName, email: senderEmail },
}
```

- The `subject` indicates the subject on the current email.

  ```
  <aura:attribute name="subject" type="String" />
  ```

- The `messageBody` indicates the email message on the current email.

  ```
  <aura:attribute name="messageBody" type="String" />
  ```

To provide the component with an event's date or location, implement the `clients:hasEventContext` interface.

```

  dates: {
          "start": value (String),
          "end": value (String),
  }
```

The Outlook and Gmail integrations don't support the following events:

- `force:navigateToList`
- `force:navigateToRelatedList`
- `force:navigateToObjectHome`
- `force:refreshView`

Note: To ensure that custom components appear correctly, enable them to adjust to variable widths.

IN THIS SECTION:

Sample Custom Components for Outlook and Gmail Integration
Review samples of custom Aura components that you can implement in the email application pane for Outlook integration and Gmail Integration.

Sample Custom Components for Outlook and Gmail Integration

Review samples of custom Aura components that you can implement in the email application pane for Outlook integration and Gmail Integration.

Apply the Selected Email Context

Here's an example of a custom Aura component you can include in your email application pane for the Outlook or Gmail integration. This component applies the context of the selected email or appointment.

```
<!-- customEmailPane.cmp -->
<aura:component implements="clients:availableForMailAppAppPage,clients:hasItemContext">

<!--
    Add these handlers to customize what happens when the attributes change
    <aura:handler name="change" value="{!v.subject}" action="{!c.handleSubjectChange}" />

    <aura:handler name="change" value="{!v.people}" action="{!c.handlePeopleChange}" />
-->

    <div id="content">
          <aura:if isTrue="{!v.mode == 'edit'}">
            You are composing the following Item: <br/>
            <aura:set attribute="else">
                You are reading the following Item: <br/>
            </aura:set>
        </aura:if>

        <h1><b>Email subject</b></h1>
        <span id="subject">{!v.subject}</span>

        <h1>To:</h1>
        <aura:iteration items="{!v.people.to}" var="to">
            {!to.name} - {!to.email} <br/>
        </aura:iteration>

        <h1>From:</h1>
        {!v.people.from.name} - {!v.people.from.email}

        <h1>CC:</h1>
        <aura:iteration items="{!v.people.cc}" var="cc">
            {!cc.name} - {!cc.email} <br/>
        </aura:iteration>

        <span class="greeting">New Email Arrived</span>, {!v.subject}!
    </div>
</aura:component>
```

Display Record Data Based On Recipients

In this example, the custom component displays account and opportunity information based on the email recipients' email addresses. The component calls a JavaScript controller function, `handlePeopleChange()`, on initialization.

```
<!-- customEmailDisplay.cmp -->
<!--
This component handles the email context on initialization.
It retrieves accounts and opportunities based on the email addresses included
in the email recipients list.
It then calculates the account and opportunity ages based on when the accounts
were created and when the opportunities will close.
```

```
-->

<aura:component
    implements="clients:availableForMailAppAppPage,clients:hasItemContext"
    controller="ComponentController">

    <aura:handler name="init" value="{!this}" action="{!c.handlePeopleChange}" />
    <aura:attribute name="accounts" type="List" />
    <aura:attribute name="opportunities" type="List" />
    <aura:iteration items="{!v.accounts}" var="acc">
            {!acc.name} => {!acc.age}
    </aura:iteration>
    <aura:iteration items="{!v.opportunities}" var="opp">
            {!opp.name} => {!opp.closesIn} Days till closing
    </aura:iteration>

</aura:component>
```

The Apex controller includes Aura-enabled methods that accept a list of emails as parameters. It queries the Contact and OpportunityContactRoles objects using SOQL and assigns the results to a List variable. You can also modify the example with your own custom objects.

```
/*ComponentController.cls*/

public class ComponentController {
    /*
    This method searches for Contacts with matching emails in the email list,
    and includes Account information in the fields. Then, it filters the
    information to return a list of objects to use on the client side.
    */
    @AuraEnabled
    public static List<Map<String, Object>> findAccountAges(List<String> emails) {
    List<Map<String, Object>> ret = new List<Map<String, Object>>();
    List<Contact> contacts = [SELECT Name, Account.Name, Account.CreatedDate
                              FROM Contact
                              WHERE Contact.Email IN :emails];
    for (Contact c: contacts) {
            Map<String, Object> item = new Map<String, Object>();
            item.put('name', c.Account.Name);
            item.put('age',
                     Date.valueOf(c.Account.CreatedDate).daysBetween(
                         System.Date.today()));
            ret.add(item);
    }
     return ret;
}

    /*
    This method searches for OpportunityContactRoles with matching emails
    in the email list.
    Then, it calculates the number of days until closing to return a list
    of objects to use on the client side.
    */
    @AuraEnabled
    public static List<Map<String, Object>> findOpportunityCloseDateTime(List<String>
emails) {
    List<Map<String, Object>> ret = new List<Map<String, Object>>();
    List<OpportunityContactRole> contacts =
           [SELECT Opportunity.Name, Opportunity.CloseDate
            FROM OpportunityContactRole
            WHERE isPrimary=true AND Contact.Email IN :emails];
    for (OpportunityContactRole c: contacts) {
            Map<String, Object> item = new Map<String, Object>();
            item.put('name', c.Opportunity.Name);
            item.put('closesIn',
                     System.Date.today().daysBetween(
                         Date.valueOf(c.Opportunity.CloseDate)));
            ret.add(item);
    }
     return ret;
  }
}
```

On component initialization, this JavaScript controller calls the helper method, `filterEmails()`, which builds a list of email addresses from the available people.

The JavaScript controller then makes a call to the server to run the actions to display information. It calls the `findOpportunityCloseDateTime` method on the Apex controller to query the opportunities days until closing. It calls the `findAccountAges` method to query the accounts ages.

Once the server returns the values, it sets the appropriate values to display on the client side.

```
/* customEmailDisplayController.js */
({
    handlePeopleChange: function(component, event, helper){
        var people = component.get("v.people");
        var peopleEmails = helper.filterEmails(people);
        var action = component.get("c.findOpportunityCloseDateTime");
        action.setParam("emails", peopleEmails);

        action.setCallback(this, function(response){
            var state = response.getState();
            if(state === "SUCCESS"){
               component.set("v.opportunities", response.getReturnValue());
            } else{
               component.set("v.opportunities",[]);
        }
});
        $A.enqueueAction(action);
        var action = component.get("c.findAccountAges");
        action.setParam("emails", peopleEmails);

        action.setCallback(this, function(response){
            var state = response.getState();
            if(state === "SUCCESS"){
               component.set("v.accounts", response.getReturnValue());
            } else{
               component.set("v.accounts",[]);
        }
});
$A.enqueueAction(action);
}
})
```

This helper function filters emails from objects.

```
/* customEmailDisplayHelper.js */({
    filterEmails : function(people){
            return this.getEmailsFromList(people.to).concat(
                this.getEmailsFromList(people.cc));
    },

    getEmailsFromList : function(list){
            var ret = [];
            for (var i in list) {
            ret.push(list[i].email);
    }
     return ret;
  }
})
```

Create Components for Forecast Pages

Create custom Aura components that are available to add to Lightning forecasts pages.

To add a custom template to a Lightning forecasts page, implement the `lightning:forecastingTemplate` interface.

To allow the component access to a Lightning forecasts page, implement the `lightning:availableForForecastingPage` interface.

Upon initialization, the component attempts to populate the following attributes to provide some forecast context. The attributes include:

- `contextPeriodIds`—The time period IDs from the user context.

  ```
  <aura:attribute name="contextPeriodIds" type="String[]" />
  ```

- `currencyIsoCode`—The ISO code of the unit of currency that the user views.

  ```
  <aura:attribute name="currencyIsoCode" type="String" />
  ```

- `forecastingOwnerId`—The forecast owner's user ID.

  ```
  <aura:attribute name="forecastingOwnerId" type="String" />
  ```

- `forecastingTerritoryId`—The forecast territory's ID.

  ```
  <aura:attribute name="forecastingTerritoryId" type="String" />
  ```

- `forecastingTypeId`—The user's selected forecast type.

  ```
  <aura:attribute name="forecastingTypeId" type="String" />
  ```

- `scope`—The supported scope type. If `scope` is undefined or null, a single territory or role page is loaded.

  ```
  <aura:attribute name="scope" type="String" />
  ```

  ```
  SupportedScopeType  : {'my_territory'}
  ```

Lightning forecasts pages don't support any standard or custom events published from custom components.

When the Lightning forecasts page changes, such as owner or forecast type, the page header publishes a Lightning message. You can subscribe to the `lightning__forecasting_flexipageUpdated` LightningMessageChannel to update your custom components based on Lightning forecasts page header changes.

Note: To ensure that custom components appear correctly, enable them to adjust to variable widths.

IN THIS SECTION:

Sample Custom Components for Forecasts Pages

Review samples of custom Aura components that you can implement in Lightning forecasts pages. Lightning forecasts pages don't support any standard or custom events published from custom components.

SEE ALSO:

Communicating Across the DOM with Lightning Message Service

Lightning Web Components Dev Guide: Subscribe and Unsubscribe from a Message Channel

Aura Component Reference: Message Channel

Sample Custom Components for Forecasts Pages

Review samples of custom Aura components that you can implement in Lightning forecasts pages. Lightning forecasts pages don't support any standard or custom events published from custom components.

Here's an example of a custom Aura component you can include in your Lightning forecasts page. To appear on the page, custom Aura components implement `lightning:availableForForecastingPage`. This component applies the context of the selected forecast.

```
<aura:component implements="lightning:availableForForecastingPage">
    <lightning:messageChannel type="lightning__forecasting_flexipageUpdated"
            onMessage="{!c.handleMessage}" scope="APPLICATION"/>

    <aura:attribute name="ownerId" type="string" default="owner"/>
    <aura:attribute name="forecastingTypeId" type="string" />
    <aura:attribute name="user" type="Object"/>
    <aura:attribute name="forecast" type="Object"/>
    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <force:recordData aura:id="recordLoader"
    recordId="{!v.ownerId}"
    fields="Name, Email, Phone, Title"
    targetFields="{!v.user}"
    />

     <force:recordData aura:id="recordLoader2"
    recordId="{!v.forecastingTypeId}"
```

```
    fields="DateType, RoleType, MasterLabel, IsAmount, IsActive"
    targetFields="{!v.forecast}"
    />

    <div>
        <lightning:card iconName="standard:user" title="{!v.user.Name}" >
            <div class="slds-p-horizontal--small">
                <p class="slds-truncate">Email : <lightning:formattedText title="Email"
value="{!v.user.Email}" /></p>
                <p class="slds-truncate">Phone : <lightning:formattedText title="Phone"
value="{!v.user.Phone}" /></p>
                <p class="slds-truncate">Title : <lightning:formattedText title="Title"
value="{!v.user.Title}" /></p>
            </div>
        </lightning:card>
    </div>

    <div>
        <lightning:card iconName="standard:forecasts" title="{!v.forecast.MasterLabel}" >

            <div class="slds-p-horizontal--small">
                <p class="slds-truncate">Role Type : <lightning:formattedText
aura:id="roleType" title="RoleType" value="{!v.forecast.RoleType}" /></p>
                <p class="slds-truncate">Date Type : <lightning:formattedText
title="DateType" value="{!v.forecast.DateType}" /></p>
            </div>
        </lightning:card>
    </div>
</aura:component>
```

The component calls a JavaScript controller function, `handleMessage()`, on initialization.

```
({
/*
This JavaScript controller is called on component initialization.
It makes a call to the server to run the actions to display information.
After the server returns the values, it sets the appropriate values to display
on the client side.
*/
    handleMessage : function(cmp, message, helper) {
        // Read the message argument to get the values in the message payload
        cmp.set("v.ownerId", message.getParam("forecastingOwnerId"));
        cmp.set("v.forecastingTypeId", message.getParam("forecastingTypeId"));
        var record = cmp.find("recordLoader");
        record.set("v.recordId", cmp.get("v.ownerId"));
        record.reloadRecord();
        var record2 = cmp.find("recordLoader2");
        record2.set("v.recordId", cmp.get("v.forecastingTypeId"));
        record2.reloadRecord();
    }
})
```

SEE ALSO:

Lightning Web Components Dev Guide: Use Components in Salesforce Targets

Create Dynamic Picklists for Your Custom Components

You can expose a component property as a picklist when the component is configured in the Lightning App Builder. The picklist's values are provided by an Apex class that you create.

For example, let's say you're creating a component for the Home page to display a custom Company Announcement record. You can use an Apex class to put the titles of all Company Announcement records in a picklist in the component's properties in the Lightning App Builder. Then, when admins add the component to a Home page, they can easily select the appropriate announcement to place on the page.

First, create a custom Apex class to use as a datasource for the picklist. The Apex class must extend the `VisualEditor.DynamicPickList` abstract class. Then add an attribute to your design file that specifies your custom Apex class as the datasource.

Here's a simple example.

Create an Apex Class

```
global class MyCustomPickList extends VisualEditor.DynamicPickList{

    global override VisualEditor.DataRow getDefaultValue(){
        VisualEditor.DataRow defaultValue = new VisualEditor.DataRow('red', 'RED');
        return defaultValue;
    }
    global override VisualEditor.DynamicPickListRows getValues() {
        VisualEditor.DataRow value1 = new VisualEditor.DataRow('red', 'RED');
        VisualEditor.DataRow value2 = new VisualEditor.DataRow('yellow', 'YELLOW');
       VisualEditor.DynamicPickListRows  myValues = new VisualEditor.DynamicPickListRows();

        myValues.addRow(value1);
        myValues.addRow(value2);
        return myValues;
    }
}
```

Note: Although `VisualEditor.DataRow` allows you to specify any Object as its value, you can specify a datasource only for String attributes. The default implementation for `isValid()` and `getLabel()` assumes that the object passed in the parameter is a String for comparison.

For more information on the `VisualEditor.DynamicPickList` abstract class, see the Apex Reference Guide.

Add the Apex Class to Your Design File

To specify an Apex class as a datasource in an existing component, add the datasource property to the attribute with a value consisting of the Apex namespace and Apex class name.

```
<design:component>
        <design:attribute name="property1" datasource="apex://MyCustomPickList"/>
</design:component>
```

Dynamic Picklist Tips and Considerations

Important: If you make an Apex datasource private using `WITH USER_MODE` in the object query, use the component only on pages that users with appropriate object access permission can view. Otherwise, the component can still be visible to users who lack appropriate permission, thereby exposing a private data string.

Let's look at some scenarios. If a user doesn't have appropriate object access permission:

- Both Aura components and Lightning web components are still visible on Lightning pages
- In LWR sites in Experience Cloud, Lightning web components are still visible on private pages and to guest users on public pages
- In Aura sites in Experience Cloud, Aura components and Lightning web components aren't visible

See Securing Data in Apex Controllers.

- You can use `VisualEditor.DesignTimePageContext` to give your picklist the context of the page that the component resides on.
- Specifying the Apex datasource as public isn't respected in managed packages. If an Apex class is public and part of a managed package, it can be used as a datasource for custom components in the subscriber org.
- Profile access on the Apex class isn't respected when the Apex class is used as a datasource. If an admin's profile doesn't have access to the Apex class but does have access to the custom component, the admin sees values provided by the Apex class on the component in the Lightning App Builder.
- The `VisualEditor.DynamicPickList` method `isValid()` runs in Experience Cloud sites when the page loads. If you don't override the method in your custom Apex class that extends `VisualEditor.DynamicPickList`, the default implementation executes `getValues()` at runtime, which can cause performance degradations. To improve performance, implement a non-operational `isValid()` method.

```
global override Boolean isValid(Object attributeValue) {
   return true;
}
```

SEE ALSO:

Apex Developer Guide: DesignTimePageContext Class

Create a Custom Lightning Page Template Component

Every standard Lightning page is associated with a default template component, which defines the page's regions and what components the page includes. Custom Lightning page template components let you create page templates to fit your business needs with the structure and components that you define. Once implemented, your custom template is available in the Lightning App Builder's new page wizard for your page creators to use.

Custom Lightning page template components are supported for record pages, app pages, and Home pages. Each page type has a different interface that the template component must implement.

- `lightning:appHomeTemplate`
- `lightning:homeTemplate`
- `lightning:recordHomeTemplate`

Important: Each template component should implement only one template interface. Template components shouldn't implement any other type of interface, such as `flexipage:availableForAllPageTypes` or `force:hasRecordId`. A template component can't multi-task as a regular component. It's either a template, or it's not.

The following steps show you how to build the template component structure for an app page, followed by adding a design resource to configure the template regions and components. After completing these steps, you can create a Lightning page in the Lightning App Builder using the Lightning page template that you configured.

1. Build the Template Component Structure

A custom template is an Aura component bundle that should include at least a .cmp resource and a design resource. The .cmp resource must implement a template interface, and declare an attribute of type `Aura.Component[]` for each template region. The `Aura.Component[]` type defines the attribute as a collection of components.

Note: The `Aura.Component[]` attribute is interpreted as a region only if it's also specified as a region in the design resource.

Here's an example of a two-column app page template .cmp resource that uses the `lightning:layout` component and the Salesforce Lightning Design System (SLDS) for styling.

When the template is viewed on a desktop, its right column takes up 30% (4 SLDS columns). The left column takes up the remaining 70% of the page width. On non-desktop form factors, the columns display as 50/50.

```
<aura:component implements="lightning:appHomeTemplate" description="Main column
 and right sidebar. On a phone, the regions are of equal width">
    <aura:attribute name="left" type="Aura.Component[]" />
    <aura:attribute name="right" type="Aura.Component[]" />

    <div>
        <lightning:layout>
            <lightning:layoutItem flexibility="grow"
                                  class="slds-m-right_small">
                {!v.left}
            </lightning:layoutItem>
            <lightning:layoutItem size="{! $Browser.isDesktop ? '4' : '6' }"
                                  class="slds-m-left_small">
                {!v.right}
            </lightning:layoutItem>
        </lightning:layout>
    </div>

</aura:component>
```

The `description` attribute on the `aura:component` tag is optional, but recommended. If you define a description, it displays as the template description beneath the template image in the Lightning App Builder new page wizard.

2. Configure Template Regions and Components in the Design Resource

The design resource controls what kind of page can be built on the template. The design resource specifies:

- What regions a page that uses the template must have.
- What kinds of components can be put into the page's regions.
- How much space the component takes on the page based on the type of device that it renders on.
- What device form factors the component supports.

Regions inherit the interface assignments that you set for the overall page, as set in the .cmp resource.

Specify regions and components using these tags:

`flexipage:template`
This tag has no attributes and acts as a wrapper for the `flexipage:region` tag. Text literals are not allowed.

`flexipage:region`
This tag defines a region on the template and has these attributes. Text literals are not allowed.

Attribute	Description
`name`	The name of an attribute in the .cmp resource marked type `Aura.Component[]`. Flags the attribute as a region.
`label`	The label of your region. This label appears in the template switching wizard in the Lightning App Builder when users map region content to a new template.
`defaultWidth`	Specifies the default width of the region. This attribute is required for all regions. Valid values are: `Small`, `Medium`, `Large`, and `Xlarge`.

`flexipage:formfactor`
Use this tag to specify how much space the component takes on the page based on the type of device that it renders on. Add this tag as a child of the `flexipage:region` tag. Use multiple `flexipage:formfactor` tags per `flexipage:region` to define flexible behavior across form factors.

Attribute	Description
`type`	The type of form factor or device the template renders on, such as a desktop or tablet. Valid values are: `Medium` (tablet), and `Large` (desktop). Because the only reasonable width value for a `Small` form factor (phone) is `Small`, you don't have to specify a `Small` type. Salesforce takes care of that association automatically.
`width`	The available size of the area that the component in this region has to render in. Valid values are: `Small`, `Medium`, `Large`, and `Xlarge`.

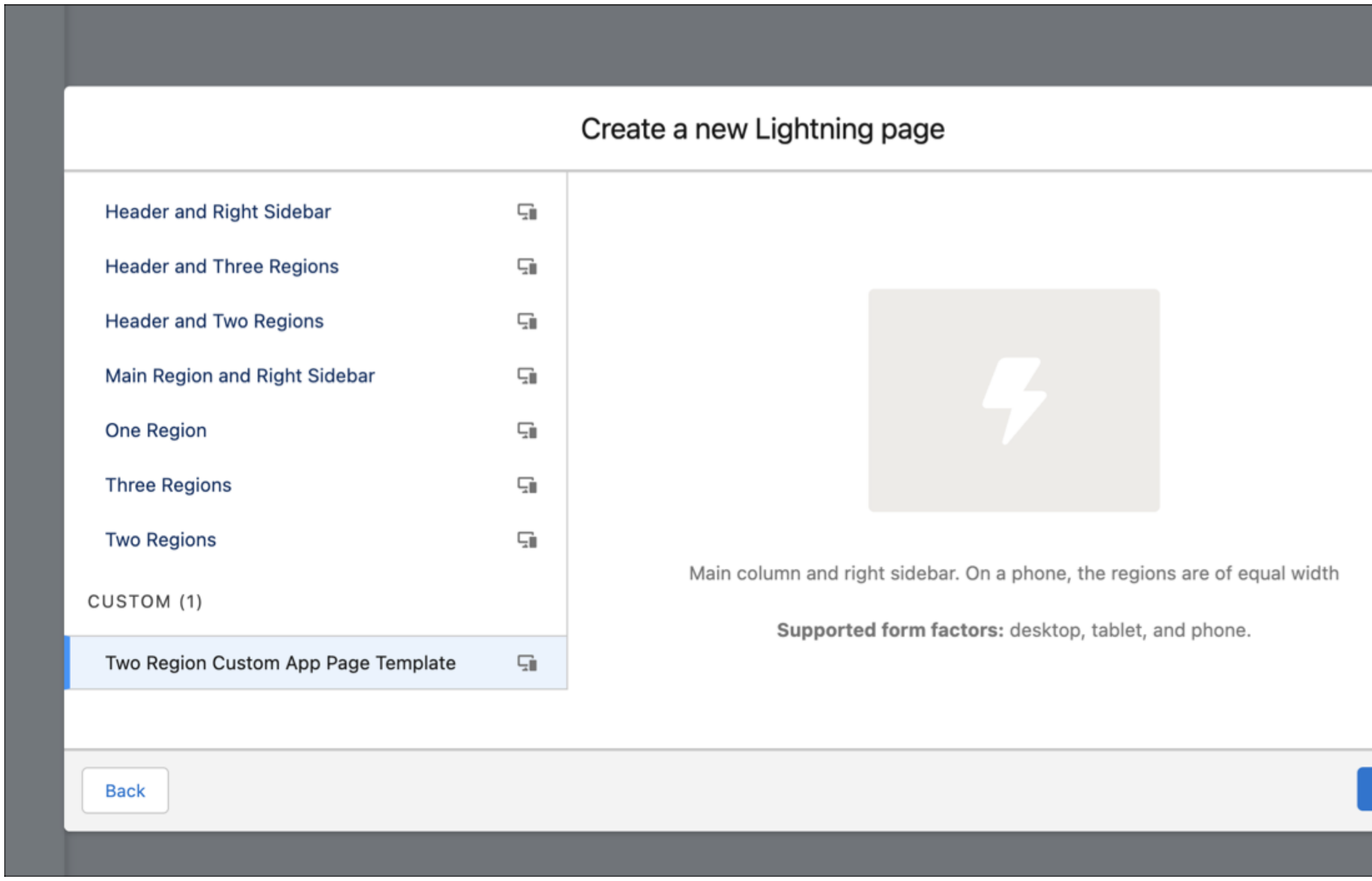
For example, in this code snippet, the region has a large width to render in when the template is rendered on a large form factor. The region has a small width to render in when the template is rendered on a medium form factor.

```
<flexipage:region name="right" label="Right Region" defaultWidth="Large">
    <flexipage:formfactor type="Large" width="Large" />
    <flexipage:formfactor type="Medium" width="Small" />
</flexipage:region>
```

Tip: You can use the `lightning:flexipageRegionInfo` subcomponent to pass region width information to a component. Doing so lets you configure your page components to render differently based on what size region they display in.

Here's the design file that goes with the sample .cmp resource. The label text in the design file displays as the name of the template in the new page wizard.

```
<design:component label="Two Region Custom App Page Template">
    <flexipage:template >
        <!-- The default width for the "left" region is "MEDIUM". In tablets,
        the width is "SMALL" -->
            <flexipage:region name="left" label="Left Region" defaultWidth="MEDIUM">
                <flexipage:formfactor type="MEDIUM" width="SMALL" />
            </flexipage:region>
            <flexipage:region name="right" label="Right Region" defaultWidth="SMALL" />
       </flexipage:template>
</design:component>
```


Create a new Lightning page

Header and Right Sidebar
Header and Three Regions
Header and Two Regions
Main Region and Right Sidebar
One Region
Three Regions
Two Regions
CUSTOM (1)
Two Region Custom App Page Template

Main column and right sidebar. On a phone, the regions are of equal width

Supported form factors: desktop, tablet, and phone.

Specify supported devices for an app or record page template component with the `<design:suppportedFormFactors>` tag set. When you create a custom template component for an app page, you must assign it both the `Large` (desktop) and `Small` (phone) form factors.

Note: Home pages support the Large form factor only.

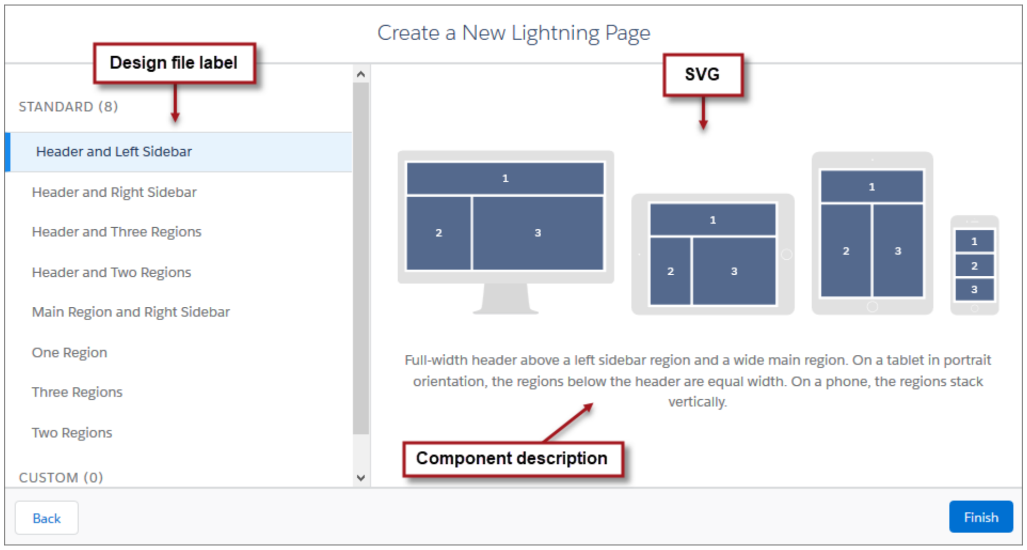
Here's the same app page template design file, with support configured for both desktop and phone.

```
<design:component label="Two Region Custom App Page Template">
    <flexipage:template >
        <!-- The default width for the "left" region is "MEDIUM". In tablets,
        the width is "SMALL" -->
            <flexipage:region name="left" label="Left Region" defaultWidth="MEDIUM">
                <flexipage:formfactor type="MEDIUM" width="SMALL" />
            </flexipage:region>
            <flexipage:region name="right" label="Right Region" defaultWidth="SMALL" />
       </flexipage:template>
       <design:supportedFormFactors>
        <design:supportedFormFactor type="Small"/>
        <design:supportedFormFactor type="Large"/>
    </design:supportedFormFactors>
</design:component>
```

3. (Optional) Add a Template Image

If you added a description to your .cmp resource, both it and the template image display when a user selects your template in the Lightning App Builder new page wizard.

You can use an SVG resource to define the custom template image.



Full-width header above a left sidebar region and a wide main region. On a tablet in portrait orientation, the regions below the header are equal width. On a phone, the regions stack vertically.

We recommend that your SVG resource is no larger than 150 KB, and no more than 160 px high and 560 px wide.

SEE ALSO:

Aura Component Bundle Design Resources

Lightning Page Template Component Best Practices

Make Your Lightning Page Components Width-Aware with lightning:flexipageRegionInfo

Lightning Page Template Component Best Practices

Keep these best practices and limitations in mind when creating Lightning page template components.

- Don't add custom background styling to a template component. It interferes with Salesforce's Lightning Experience page themes.
- We strongly recommend including supported form factor information in the design file of all of your components. If you don't, the component might behave in unexpected ways.
- Template component supported form factors must be equal to, or a subset of, the supported form factors of its page type.
- Once a component is in use on a Lightning page, you can only increase the supported form factors for the component, not decrease them.
- Including scrolling regions in your template component can cause problems when you try to view it in the Lightning App Builder.
- Custom templates can't be extensible nor extended—you can't extend a template from anything else, nor can you extend other things from a template.
- Using getters to get the regions as variables works at design time but not at run time. Here's an example of what we mean.

```
<aura:component implements="lightning:appHomeTemplate">
    <aura:attribute name="region" type="Aura.Component[]" />
    <aura:handler name="init" value="{!this}" action="{!c.init}" />

    <div>
        {!v.region}
    </div>

</aura:component>
```

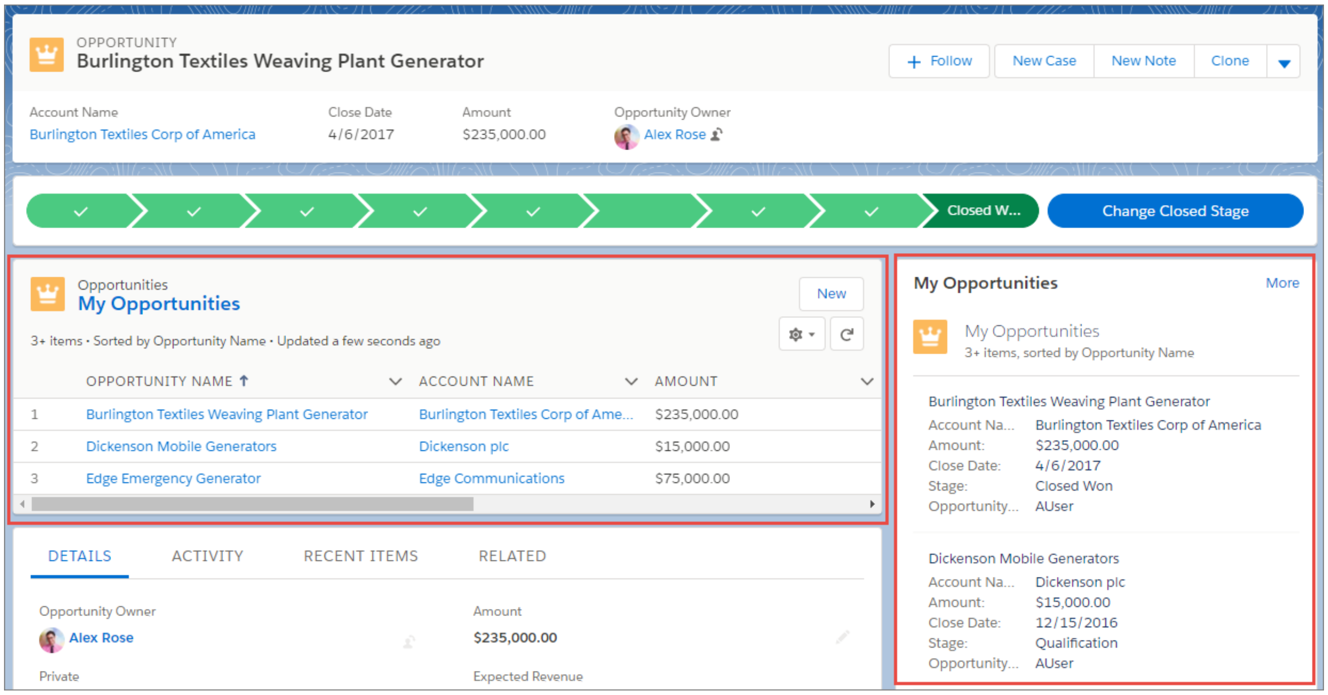
```
{
    init : function(component, event, helper) {
        var region = cmp.get('v.region'); // This will fail at run time.
        ...
    }
}
```

- You can remove regions from a template if it's not being used by a Lightning page, and if it's not set to access=global. You can add regions at any time.
- A region can be used more than once in the code, but only one instance of the region should render at run time.
- A template component can contain up to 25 regions.
- The order that you list the regions in a page template is the order that the regions appear in when admins migrate region content using the template switching wizard in the Lightning App Builder. We recommend that you label the regions and list them in a logical order in your template, such as top to bottom or left to right.

Make Your Lightning Page Components Width-Aware with `lightning:flexipageRegionInfo`

When you add a component to a region on a page in the Lightning App Builder, the `lightning:flexipageRegionInfo` sub-component passes the width of that region to its parent component. With `lightning:flexipageRegionInfo` and some strategic CSS, you can tell the parent component to render in different ways in different regions at runtime.

For example, the List View component renders differently in a large region than it does in a small region as it's a width-aware component.


OPPORTUNITY
Burlington Textiles Weaving Plant Generator
Follow
New Case
New Note
Clone
Account Name
Burlington Textiles Corp of America
Close Date
4/6/2017
Amount
$235,000.00
Opportunity Owner
Alex Rose
Closed W...
Change Closed Stage
Opportunities
My Opportunities
New
3+ items • Sorted by Opportunity Name • Updated a few seconds ago
OPPORTUNITY NAME
ACCOUNT NAME
AMOUNT
1 Burlington Textiles Weaving Plant Generator Burlington Textiles Corp of Ame... $235,000.00
2 Dickenson Mobile Generators Dickenson plc $15,000.00
3 Edge Emergency Generator Edge Communications $75,000.00
DETAILS
ACTIVITY
RECENT ITEMS
RELATED
Opportunity Owner
Alex Rose
Amount
$235,000.00
Private
Expected Revenue
My Opportunities
More
My Opportunities
3+ items, sorted by Opportunity Name
Burlington Textiles Weaving Plant Generator
Account Na... Burlington Textiles Corp of America
Amount: $235,000.00
Close Date: 4/6/2017
Stage: Closed Won
Opportunity... AUser
Dickenson Mobile Generators
Account Na... Dickenson plc
Amount: $15,000.00
Close Date: 12/15/2016
Stage: Qualification
Opportunity... AUser

Valid region width values are: `Small`, `Medium`, `Large`, and `Xlarge`.

You can use CSS to style your component and to help determine how your component renders. Here's an example.

This simple component has two fields, field1 and field2. The component renders with the fields side by side, filling 50% of the region's available width when not in a small region. When the component is in a small region, the fields render as a list, using 100% of the region's width.

```
<aura:component implements="flexipage:availableForAllPageTypes">
    <aura:attribute name="width" type="String"/>
    <lightning:flexipageRegionInfo width="{!v.width}"/>
    <div class="{! 'container' + (v.width=='SMALL'?' narrowRegion':'')}">
        <div class="{! 'eachField f1' + (v.width=='SMALL'?' narrowRegion':'')}">
            <lightning:input name="field1" label="First Name"/>
        </div>
        <div class="{! 'eachField f2' + (v.width=='SMALL'?' narrowRegion':'')}">
            <lightning:input name="field2" label="Last Name"/>
        </div>
    </div>
</aura:component>
```

Here's the CSS file that goes with the component.

```
.THIS .eachField.narrowRegion{
    width:100%;
}
.THIS .eachField{
    width:50%;
    display:inline-block;
}
```

Tips and Considerations for Configuring Components for Lightning Pages and the Lightning App Builder

Keep these guidelines in mind when creating components and component bundles for Lightning pages and the Lightning App Builder.

Note: Mark your resources, such as a component, with `access="global"` to make the resource usable outside of your own org. For example, if you want a component to be usable in an installed package or by a Lightning App Builder user or a Experience Builder user in another org.

You can also create documentation for a component, event, or interface marked `access="global"`. This documentation is automatically displayed in the Component Library of an org that uses or installs your package.

Components

- Set a friendly name for the component using the `label` attribute in the element in the design file, such as `<design:component label="foo">`.
- Make your components fill 100% of the width (including margins) of the region that they display in.
- Don't set absolute width values on your components.
- If components require interaction, they must provide an appropriate placeholder behavior in declarative tools.
- A component must never display a blank box. Think of how other sites work. For example, Facebook displays an outline of the feed before the actual feed items come back from the server. The outline improves the user's perception of UI responsiveness.
- If the component depends on a fired event, then give it a default state that displays before the event fires.
- Style components in a manner consistent with the styling of Lightning Experience and consistent with the Salesforce Design System.
- The Lightning App Builder manages spacing between components automatically. Don't add margins to your component CSS, and avoid adding padding.
- Don't use `float` or `position: absolute` in your CSS properties. These properties break the component out of the page structure and, as a result, break the page.

Attributes

- Use the design file to control which attributes are exposed to the Lightning App Builder.
- Make your attributes easy to use and understandable to an administrator. Don't expose SOQL queries, JSON objects, or Apex class names.
- Give your required attributes default values. When a component that has required attributes with no default values is added to the App Builder, it appears invalid, which is a poor user experience.
- Use basic supported types (string, integer, boolean) for any exposed attributes.
- Specify a min and max attribute for integer attributes in the `<design:attribute>` element to control the range of accepted values.
- String attributes can provide a data source with a set of predefined values allowing the attribute to expose its configuration as a picklist.
- Give all attributes a label with a friendly display name.
- Provide descriptions to explain the expected data and any guidelines, such as data format or expected range of values. Description text appears as a tooltip in the Property Editor.
- To delete a design attribute for a component that implements the `flexipage:availableForAllPageTypes` or `forceCommunity:availableForAllPageTypes` interface, first remove the interface from the component before

deleting the design attribute. Then reimplement the interface. If the component is referenced in a Lightning page, you must remove the component from the page before you can change it.

Limitations

- The Lightning App Builder doesn't support the Map, Object, or java:// complex types.
- When you use the Lightning App Builder, there's a known limitation when you edit a group page. Your changes appear when you visit the group from the Groups tab. Your changes don't appear when you visit the group from the Recent Groups list on the Chatter tab.
- Custom components that serve as containers, such as custom Tabs or Accordion components, aren't supported in Lightning App Builder. They display on the canvas, but you can't interact with them or put any components inside them.

SEE ALSO:

Configure Components for Lightning Pages and the Lightning App Builder

Configure Components for Lightning Experience Record Pages

Use Aura Components in Experience Builder

To use a custom Aura component in Experience Builder, you must configure the component and its component bundle so that they're compatible.

Note: As of Spring '21, you can build Experience Builder sites using two programming models: the Lightning Web Components model, and the original Aura Components model. The Marketing Website template is based on LWC and can only be used with Lightning web components, not Aura components. Other templates are based on the Aura Components model and can use both Lightning web components and Aura components. See the Experience Builder Developer Guide for more information.

IN THIS SECTION:

Configure Components for Experience Builder

Make your custom Aura components available to drag to the Lightning Components pane in Experience Builder.

Create Custom Theme Layout Components for Experience Builder

Create a custom theme layout to transform the appearance and overall structure of the pages in the Customer Service template.

Create Custom Component for Guest User Flows

Allow flows for your Experience Cloud guest users to provide alternative user registration screens, complex decision trees, and conditional forms to gather user information. The following example uses the Site Class API. For more information, see "Site Class" in the Salesforce Apex Developer Guide.

Create Custom Search and Profile Menu Components for Experience Builder

Create custom components to replace the Customer Service template's standard Profile Header and Search & Post Publisher components in Experience Builder.

Create Custom Content Layout Components for Experience Builder

Experience Builder includes several ready-to-use layouts that define the content regions of your page, such as a two-column layout with a 2:1 ratio. However, if you need a layout that's customized for your site, create a custom content layout component to use when building new pages in Experience Builder. You can also update the content layout of the default pages that come with your site template.

Configure Components for Experience Builder

Make your custom Aura components available to drag to the Lightning Components pane in Experience Builder.

Note: As of Spring '21, you can build Experience Builder sites using two programming models: the Lightning Web Components model, and the original Aura Components model. The Marketing Website template is based on LWC and can only be used with Lightning web components, not Aura components. Other templates are based on the Aura Components model and can use both Lightning web components and Aura components. See the Experience Builder Developer Guide for more information.

Add a New Interface to Your Component

To appear in Experience Builder, a component must implement the `forceCommunity:availableForAllPageTypes` interface.

Here's the sample code for a simple "Hello World" component.

```
<aura:component implements="forceCommunity:availableForAllPageTypes" access="global">
    <aura:attribute name="greeting" type="String" default="Hello" access="global" />
    <aura:attribute name="subject" type="String" default="World" access="global" />

    <div style="box">
      <span class="greeting">{!v.greeting}</span>, {!v.subject}!
    </div>
</aura:component>
```

Note: Mark your resources, such as a component, with `access="global"` to make the resource usable outside of your own org. For example, if you want a component to be usable in an installed package or by a Lightning App Builder user or a Experience Builder user in another org.

You can also create documentation for a component, event, or interface marked `access="global"`. This documentation is automatically displayed in the Component Library of an org that uses or installs your package.

Next, add a design resource to your component bundle. A design resource describes the design-time behavior of an Aura component—information that visual tools need to allow adding the component to a page or app. It contains attributes that are available for administrators to edit in Experience Builder.

Adding this resource is similar to adding it for the Lightning App Builder. For more information, see Configure Components for Lightning Pages and the Lightning App Builder.

Important: When you add custom components to your Experience Builder site, they can bypass the object- and field-level security (FLS) you set for the guest user profile. Lightning components don't automatically enforce CRUD and FLS when referencing objects or retrieving the objects from an Apex controller. This means that the framework continues to display records and fields for which users don't have CRUD permissions and FLS visibility. You must manually enforce CRUD and FLS in your Apex controllers. Alternatively, use a base component that implements Lightning Data Service on page 384.

SEE ALSO:

Component Bundles

Standard Design Tokens for Experience Builder Sites

Create Custom Theme Layout Components for Experience Builder

Create a custom theme layout to transform the appearance and overall structure of the pages in the Customer Service template.

A theme layout component is the top-level layout for the template pages in your site. Theme layout components are organized and applied to your pages through theme layouts. A theme layout component includes the common header and footer, and often includes navigation, search, and the user profile menu. In contrast, the content layout defines the content regions of your pages. The next image shows a two-column content layout.

A theme layout type categorizes the pages in your Experience Builder site that share the same theme layout.

When you create a custom theme layout component in the Developer Console, it appears in Experience Builder in the **Settings** > **Theme** area. Here you can assign it to new or existing theme layout types. Then you apply the theme layout type—and then the theme layout—in the page's properties.

1. Add an Interface to Your Theme Layout Component

A theme layout component must implement the `forceCommunity:themeLayout` interface to appear in Experience Builder in the **Settings** > **Theme** area.

Explicitly declare `{!v.body}` in your code to ensure that your theme layout includes the content layout. Add `{!v.body}` wherever you want the page's contents to appear within the theme layout.

You can add components to the regions in your markup or leave regions open for users to drag-and-drop components into. Attributes declared as `Aura.Component[]` and included in your markup are rendered as open regions in the theme layout that users can add components to.

In Customer Service, the Template Header consists of these locked regions:

- `search`, which contains the Search Publisher component
- `profileMenu`, which contains the User Profile Menu component
- `navBar`, which contains the Navigation Menu component

To create a custom theme layout that reuses the existing components in the Template Header region, declare `search`, `profileMenu`, or `navBar` as the attribute name value, as appropriate. For example:

```
<aura:attribute name="navBar" type="Aura.Component[]" required="false" />
```

Tip: If you create a custom profile menu or a search component, declaring the attribute name value also lets users select the custom component when using your theme layout.

Here's the sample code for a simple theme layout.

```
<aura:component implements="forceCommunity:themeLayout" access="global" description="Sample
 Custom Theme Layout">
    <aura:attribute name="search" type="Aura.Component[]" required="false"/>
    <aura:attribute name="profileMenu" type="Aura.Component[]" required="false"/>
    <aura:attribute name="navBar" type="Aura.Component[]" required="false"/>
    <aura:attribute name="newHeader" type="Aura.Component[]" required="false"/>
    <div>
        <div class="searchRegion">
            {!v.search}
        </div>
        <div class="profileMenuRegion">
            {!v.profileMenu}
        </div>
        <div class="navigation">
            {!v.navBar}
        </div>
        <div class="newHeader">
```

```
            {!v.newHeader}
        </div>
        <div class="mainContentArea">
            {!v.body}
        </div>
    </div>
</aura:component>
```

Note: Mark your resources, such as a component, with `access="global"` to make the resource usable outside of your own org. For example, if you want a component to be usable in an installed package or by a Lightning App Builder user or a Experience Builder user in another org.

You can also create documentation for a component, event, or interface marked `access="global"`. This documentation is automatically displayed in the Component Library of an org that uses or installs your package.

Note: If you want to use a new, customizable profile menu instead of a self-service profile menu, you must declare the themeHeaderProfileMenu attribute instead of profileMenu in the theme layout component. This only works in a B2B store or where an out-of-box theme has been applied.

2. Add a Design Resource to Include Theme Properties

You can expose theme layout properties in Experience Builder by adding a design resource to your bundle.

This example adds two checkboxes to a theme layout called Small Header.

```
<design:component label="Small Header">
    <design:attribute name="blueBackground" label="Blue Background"/>
    <design:attribute name="smallLogo" label="Small Logo"/>
</design:component>
```

The design resource only exposes the properties. Implement the properties in the component.

```
<aura:component implements="forceCommunity:themeLayout" access="global" description="Small
 Header">
    <aura:attribute name="blueBackground" type="Boolean" default="false"/>
    <aura:attribute name="smallLogo" type="Boolean" default="false" />
    ...
```

Design resources must be named *`componentName`*`.design`.

3. Add a CSS Resource to Avoid Overlapping Issues

Add a CSS resource to your bundle to style the theme layout as needed.

To avoid overlapping issues with positioned elements, such as dialog boxes or hovers:

- Apply CSS styles.

```
.THIS {
    position: relative;
    z-index: 1;
}
```

- Wrap the elements in your custom theme layout in a `div` tag.

```
<div class="mainContentArea">
    {!v.body}
</div>
```

Note: For custom theme layouts, SLDS is loaded by default.

CSS resources must be named *`componentName`*`.css`.

SEE ALSO:

Create Custom Search and Profile Menu Components for Experience Builder

Salesforce Help: Custom Theme Layouts and Theme Layout Types

Create Custom Component for Guest User Flows

Allow flows for your Experience Cloud guest users to provide alternative user registration screens, complex decision trees, and conditional forms to gather user information. The following example uses the Site Class API. For more information, see “Site Class” in the Salesforce Apex Developer Guide.

1. Create a Custom Aura Component

Using Guest User Flows for login or self registration requires a custom component that implements `lightning:availableForFlowScreens`.

Here’s the sample code for a simple data collection preferences flow.

```
<aura:component implements="lightning:availableForFlowScreens"
controller="CommunitySelfRegController">
    <aura:attribute name="email" type="String" default=""/>
    <aura:attribute name="fname" type="String" default=""/>
    <aura:attribute name="lname" type="String" default=""/>
    <aura:attribute name="starturl" type="String" default=""/>
    <aura:attribute name="password" type="String" default=""/>
    <aura:attribute name="hasOptedTracking" type="Boolean" default="false"/>
    <aura:attribute name="hasOptedSolicit" type="Boolean" default="false"/>
    <aura:attribute name="op_url" type="String" default="" description="login url after
user is created. "/>

    <aura:handler name="init" value="{!this}" action="{!c.init}" />

    <aura:if isTrue="{! (empty(v.op_url))}">
        <!-- empty url, the user is not yet created  -->
        <h3> Registering user. Please wait. </h3>

        <aura:set attribute="else">
            <!-- User created, show link to login -->
            <h3> Success! Your account has been created. </h3>

            <button class="slds-button slds-button_neutral"
onclick="{!c.login}">Login</button>
        </aura:set>
    </aura:if>
</aura:component>
```

Controller file:

```
({
    init : function(cmp) {
        let email = cmp.get("v.email"),
            fname = cmp.get("v.fname"),
            lname = cmp.get("v.lname"),
            pass = cmp.get("v.password"),
            startUrl = cmp.get("v.starturl"),
            hasOptedSolicit = cmp.get("v.hasOptedSolicit"),
            hasOptedTracking = cmp.get("v.hasOptedTracking");

        let action = cmp.get("c.createExternalUser");
        action.setParams(
            {
                username: email,
                password: pass,
                startUrl: startUrl,
                fname: fname,
                lname: lname,
                hasOptedTracking: hasOptedTracking,
                hasOptedSolicit: hasOptedSolicit
            });

        action.setCallback(this, function(res) {
            if (action.getState() === "SUCCESS") {
                cmp.set("v.op_url", res.getReturnValue());
            }
        });
        $A.enqueueAction(action);
    },

    login: function(cmp){
        let url = cmp.get("v.op_url");
        window.location.href = url;
    }
})
```

Design file:

```
<design:component>
    <design:attribute name="email" />
    <design:attribute name="fname"  />
    <design:attribute name="lname" />
    <design:attribute name="password" />
    <design:attribute name="hasOptedTracking" />
    <design:attribute name="hasOptedSolicit" />
</design:component>
```

2. Create an Apex Class

The following example creates a class, `CommunitySelfRegController`, which is used with your Aura component to register new Experience Cloud site users.

Note: Adding self registration with a flow requires the following:

- The `UserPreferencesHideS1BrowserUI` preference should be set to True. This prevents the mobile UI from defaulting to the Salesforce Mobile App interface rather than your Experience Builder site.
- `CommunityNickname` is required and must be a unique value.
- The self registration preference should be enabled in your site with a valid profile and account.

```
public class CommunitySelfRegController {
    @AuraEnabled
    public static String createExternalUser(
        String username, String password, String startUrl, String fname,
        String lname, Boolean hasOptedTracking, Boolean hasOptedSolicit) {
            Savepoint sp = null;
            try {
                sp = Database.setsavepoint();
                system.debug(sp);

                // Creating a user object.
                User u = new User();
                u.Username = username;
                u.Email = username;
                u.FirstName = fname;
                u.LastName = lname;

                // Default UI for mobile is set to S1 for user created using site object.

                // Enable this perm to change it to community (Experience Cloud).
                u.UserPreferencesHideS1BrowserUI = true;

                // Generating unique value for Experience Cloud nickname.
    String nickname = ((fname != null && fname.length() > 0) ? fname.substring(0,1) : ''
) + lname.substring(0,1);
             nickname += String.valueOf(Crypto.getRandomInteger()).substring(1,7);
                u.CommunityNickname = nickname;

                System.debug('creating user');

                // Creating portal user.
                // Passing in null account ID forces the system to read this from the
network setting (set using Experience Workspaces).
                String userId = Site.createPortalUser(u, null, password);

                // Setting consent selection values.
                // For this, GDPR (Individual and Consent Management) needs to be enabled
 in the org.
                Individual ind = new Individual();
                ind.LastName = lname;
                ind.HasOptedOutSolicit = !hasOptedSolicit;
                ind.HasOptedOutTracking = !hasOptedTracking;
                insert(ind);

                // Other contact information can be updated here.
                Contact contact = new Contact();
                contact.Id = u.ContactId;
                contact.IndividualId = ind.Id;
                update(contact);

                // return login url.
                if (userId != null && password != null && password.length() > 1) {
                  ApexPages.PageReference lgn = Site.login(username, password, startUrl);

                    return lgn.getUrl();
                }
            }
            catch (Exception ex) {
                Database.rollback(sp);
                System.debug(ex.getMessage());
                return null;
            }
            return null;
        }
}
Collapse

}
```

SEE ALSO:

Salesforce Help: Allow Guest Users to Access Flows

Create Custom Search and Profile Menu Components for Experience Builder

Create custom components to replace the Customer Service template's standard Profile Header and Search & Post Publisher components in Experience Builder.

`forceCommunity:profileMenuInterface`

Add the `forceCommunity:profileMenuInterface` interface to an Aura component to allow it to be used as a custom profile menu component for the Customer Service site template. After you create a custom profile menu component, admins can select it in Experience Builder in **Settings** > **Theme** to replace the template's standard Profile Header component.

Here's the sample code for a simple profile menu component.

```
<aura:component implements="forceCommunity:profileMenuInterface" access="global">
    <aura:attribute name="options" type="String[]" default="['Option 1', 'Option 2']"/>
    <lightning:avatar variant="circle" src="" fallbackIconName="standard:person_account"
alternativeText="Account User"/>
    <lightning:buttonMenu alternativeText="Profile Menu" variant="container"
iconName="utility:connected_apps">
        <aura:iteration items="{!v.options}" var="itemLabel">
            <lightning:menuItem label="{!itemLabel}" />
        </aura:iteration>
    </lightning:buttonMenu>
</aura:component>
```

`forceCommunity:searchInterface`

Add the `forceCommunity:searchInterface` interface to an Aura component to allow it to be used as a custom search component for the Customer Service site template. After you create a custom search component, admins can select it in Experience Builder in **Settings** > **Theme** to replace the template's standard Search & Post Publisher component.

Here's the sample code for a simple search component.

```
<aura:component implements="forceCommunity:searchInterface" access="global">
    <div onkeyup="{! c.handleKeyUp }">
    <lightning:input
            aura:id="search-input"
            label="Search"
            type="search"
            variant="label-hidden"
        />
    </div>
</aura:component>
```

```
({
    handleKeyUp: function (cmp, evt) {
        var isEnterKey = evt.keyCode === 13;
        if (isEnterKey) {
            var queryTerm = cmp.find('search-input').get('v.value');
            //do something with user input
        }
    }
})
```

SEE ALSO:

Create Custom Theme Layout Components for Experience Builder

Salesforce Help: Custom Theme Layouts and Theme Layout Types

Create Custom Content Layout Components for Experience Builder

Experience Builder includes several ready-to-use layouts that define the content regions of your page, such as a two-column layout with a 2:1 ratio. However, if you need a layout that's customized for your site, create a custom content layout component to use when building new pages in Experience Builder. You can also update the content layout of the default pages that come with your site template.

When you create a custom content layout component in the Developer Console, it appears in Experience Builder in the New Page and the Change Layout dialog boxes.

1. Add a New Interface to Your Content Layout Component

To appear in the New Page and the Change Layout dialog boxes in Experience Builder, a content layout component must implement the `forceCommunity:layout` interface.

Here's the sample code for a simple two-column content layout.

```
<aura:component implements="forceCommunity:layout" description=”Custom Content Layout”
access="global">
    <aura:attribute name="column1" type="Aura.Component[]" required="false"></aura:attribute>

    <aura:attribute name="column2" type="Aura.Component[]" required="false"></aura:attribute>


    <div class="container">
        <div class="contentPanel">
            <div class="left">
                {!v.column1}
            </div>
            <div class="right">
                {!v.column2}
            </div>
        </div>
    </div>
</aura:component>
```

Note: Mark your resources, such as a component, with `access="global"` to make the resource usable outside of your own org. For example, if you want a component to be usable in an installed package or by a Lightning App Builder user or a Experience Builder user in another org.

You can also create documentation for a component, event, or interface marked `access="global"`. This documentation is automatically displayed in the Component Library of an org that uses or installs your package.

2. Add a CSS Resource to Your Component Bundle

Next, add a CSS resource to style the content layout as needed.

Here's the sample CSS for our simple two-column content layout.

```

.THIS .contentPanel:before,
.THIS .contentPanel:after {
    content: " ";
    display: table;
}
.THIS .contentPanel:after {
    clear: both;
}
.THIS .left {
    float: left;
    width: 50%;
}
.THIS .right {
    float: right;
    width: 50%;
}
```

CSS resources must be named *`componentName`*`.css`.

3. Optional: Add an SVG Resource to Your Component Bundle

You can include an SVG resource in your component bundle to define a custom icon for the content layout component when it appears in the Experience Builder.

The recommended image size for a content layout component in Experience Builder is 170px by 170px. However, if the image has different dimensions, Experience Builder scales the image to fit.

SVG resources must be named `componentName.svg`.

SEE ALSO:

Component Bundles

Standard Design Tokens for Experience Builder Sites

Use Aura Components with Flows

Customize the look-and-feel and functionality of your flows by adding Lightning components to them. Or wrap a flow in an Aura component to configure the flow at runtime, such as to control how a paused flow is resumed.

IN THIS SECTION:

Considerations for Configuring Components for Flows

Before you configure an Aura component for a flow, determine whether it should be available in flow screens or as flow actions and understand how to map data types between a flow and an Aura component. Then review some considerations for defining attributes and how components behave in flows at runtime.

Customize Flow Screens Using Aura Components

To customize the look and feel of your flow screen, build a custom Aura component. Configure the component and its design resource so that they're compatible with flow screens. Then in Flow Builder, add a screen component to the screen.

Create Flow Local Actions Using Aura Components

To execute client-side logic in your flow, build or modify custom Aura components to use as local actions in flows. For example, get data from third-party systems without going through the Salesforce server, or open a URL in another browser tab. Once you configure the Aura component's markup, client-side controller, and design resource, it's available in Flow Builder as a Core Action element.

Embed a Flow in a Custom Aura Component

Once you embed a flow in an Aura component, use JavaScript and Apex code to configure the flow at run time. For example, pass values into the flow or to control what happens when the flow finishes. `lightning:flow` supports only screen flows and autolaunched flows.

Display Flow Stages with an Aura Component

If you've added stages to your flow, display them to flow users with an Aura component, such as `lightning:progressindicator`.

Considerations for Configuring Components for Flows

Before you configure an Aura component for a flow, determine whether it should be available in flow screens or as flow actions and understand how to map data types between a flow and an Aura component. Then review some considerations for defining attributes and how components behave in flows at runtime.

Note:

- Lightning components in flows must comply with Lightning Locker restrictions.
- Flows that include Lightning components are supported only in Lightning runtime.

IN THIS SECTION:

SEE ALSO:

Component Library: lightning:availableForFlowScreens Interface

Component Library: lightning:availableForFlowActions Interface

Security for Lightning Components: Lightning Locker

Flow Screen Components vs. Flow Action Components

You can make your Aura component available in flow screens or as a flow action. When choosing between the flow interfaces, consider what purpose the component serves in the flow.

For this use case...	Create a...
Provide UI for the user to interact with	Flow screen component
Update the screen in real time	Flow screen component
Prevent the flow from continuing until the component is done	Flow action component
Make direct data queries to on-premise or private cloud data	Flow action component

SEE ALSO:

Component Library: lightning:availableForFlowScreens Interface

Component Library: lightning:availableForFlowActions Interface

Which Custom Lightning Component Attribute Types Are Supported in Flows?

Not all custom Lightning component data types are supported in flows. You can map only these types and their associated collection types between flows and custom Lightning components.

Flow Data Type	Lightning Component Attribute Type	Valid Values
Apex	Custom Apex Class	Apex classes that define `@AuraEnabled` fields. Supported data types in an Apex class are Boolean, Integer, Long, Decimal, Double, Date, DateTime, and String. Single values as well as Lists are supported for each data type.
Boolean	Boolean	• True values: *`true`*, *`1`*, or equivalent expression • False values: *`false`*, *`0`*, or equivalent expression
Currency	Number	Numeric value or equivalent expression
Date	Date	*`"YYYY-MM-DD"`* or equivalent expression
Date/Time (API name is DateTime)	DateTime	*`"YYYY-MM-DDThh:mm:ssZ"`* or equivalent expression
Number	Number	Numeric value or equivalent expression
Multi-Select Picklist (API name is Multi-Select Picklist.)	String	String value or equivalent expression using this format: *`"Blue; Green; Yellow"`*
Picklist	String	String value or equivalent expression
Record, with a specified object (API name is SObject.)	The API name of the specified object, such as Account or Case	Map of key-value pairs or equivalent expression. Flow record values map only to attributes whose type is the specific object. For example, an account record variable can be mapped only to an attribute whose type is Account. Flow data types aren't compatible with attributes whose type is Object.
Text (API name is Text.)	String	String value or equivalent expression

SEE ALSO:

Component Library: lightning:flow Component

Component Library: lightning:availableForFlowScreens Interface

Component Library: lightning:availableForFlowActions Interface

Design Attribute Considerations for Flow Screen Components and Local Actions

To expose an attribute in Flow Builder, define a corresponding `design:attribute` in the component bundle's design resource. Keep these guidelines in mind when defining design attributes for flows.

Supported Attributes on `design:attribute` Nodes

In a `design:attribute` node, Flow Builder supports only the `name`, `label`, `description`, and `default` attributes. The other attributes, like `min` and `max`, are ignored.

For example, for this design attribute definition, Flow Builder ignores required and placeholder.

```
<design:attribute name="greeting" label="Greeting" placeholder="Hello" required="true"/>
```

Calculating Minimum and Maximum Values for an Attribute

To validate min and max lengths for a component attribute, use a flow formula or the component's client-side controller.

Modifying or Deleting `design:attribute` Nodes

If a component's attribute is referenced in a flow, you can't change the attribute's type or remove it from the design resource. This limitation applies to all flow versions, not just active ones. Remove references to the attribute in all flow versions, and then edit or delete the attribute in the design resource.

SEE ALSO:

Component Library: lightning:availableForFlowScreens Interface

Component Library: lightning:availableForFlowActions Interface

Runtime Considerations for Flows That Include Aura Components

Depending on where you run your flow, Aura components may look or behave differently than expected. The flow runtime app that's used for some distribution methods doesn't include all the necessary resources from the Lightning Component framework. When a flow is run from Flow Builder or a direct flow URL (https://yourDomain.my.salesforce.com/flow/MyFlowName), `force` and `lightning` events aren't handled.

To verify the behavior of your Aura components, test your flow in a way that handles `force` and `lightning` events, such as `force:showToast`. You can also add the appropriate event handlers directly to your component.

Distribution Method	Handles `force` and `lightning` Events
Direct flow URL	
Run and Debug buttons in Flow Builder	
Run links on flow detail pages and list views	
Web tab	
Custom button or link	
Lightning page	✓
Experience Builder site page	✓
Flow action	✓
Utility bar	✓
`flow:interview` Visualforce component	

Distribution Method	**Handles `force` and `lightning` Events**
`lightning:flow` Aura component	Depends on where you embed the component or whether your component includes the appropriate event handlers

SEE ALSO:

Component Library: lightning:availableForFlowScreens Interface

Component Library: lightning:availableForFlowActions Interface

Events Handled in the Salesforce Mobile App and Lightning Experience

Customize Flow Screens Using Aura Components

To customize the look and feel of your flow screen, build a custom Aura component. Configure the component and its design resource so that they're compatible with flow screens. Then in Flow Builder, add a screen component to the screen.

IN THIS SECTION:

Configure Components for Flow Screens

Make your custom Aura components available to flow screens in Flow Builder by implementing the `lightning:availableForFlowScreens` interface.

Control Flow Navigation from an Aura Component

By default, users navigate a flow by clicking standard buttons at the bottom of each screen. The `lightning:availableForFlowScreens` interface provides two attributes to help you fully customize your screen's navigation. To figure out which navigation actions are available for the screen, loop through the `availableActions` attribute. To programmatically trigger one of those actions, call the `navigateFlow` action from your JavaScript controller.

Customize the Flow Header with an Aura Component

To replace the flow header with an Aura component, use the `screenHelpText` parameter from the `lightning:availableForFlowScreens` interface.

Dynamically Update a Flow Screen with an Aura Component

To conditionally display a field on your screen, build an Aura component that uses `aura:if` to check when parts of the component should appear.

SEE ALSO:

Component Library: lightning:availableForFlowScreens Interface

Create Flow Local Actions Using Aura Components

Configure Components for Flow Screens

Make your custom Aura components available to flow screens in Flow Builder by implementing the `lightning:availableForFlowScreens` interface.

Here's the sample code for a simple "Hello World" component.

```
<aura:component implements="lightning:availableForFlowScreens" access="global">
    <aura:attribute name="greeting" type="String" access="global" />
    <aura:attribute name="subject" type="String" access="global" />

    <div style="box">
      <span class="greeting">{!v.greeting}</span>, {!v.subject}!
    </div>
</aura:component>
```

Note: Mark your resources, such as a component, with `access="global"` to make the resource usable outside of your own org. For example, you want a component to be usable in an installed package or by a Lightning App Builder user or an Experience Builder user in another org.

To make an attribute's value customizable in Flow Builder, add it to the component's design resource. That way, flow admins can pass values between that attribute and the flow when they configure the screen component.

With this sample design resource, flow admins can customize the values for the "Hello World" component's attributes.

```
<design:component label="Hello World">
   <design:attribute name="greeting" label="Greeting" />
   <design:attribute name="subject" label="Subject" />
</design:component>
```

A design resource describes the design-time behavior of a Lightning component—information that visual tools require to allow adding the component to a page or app. Adding this resource is similar to adding it for the Lightning App Builder.

When admins reference this component in a flow, they can set each attribute using values from the flow. And they can store each attribute's output value in a flow variable.

SEE ALSO:

Control Flow Navigation from an Aura Component

Component Library: lightning:availableForFlowScreens Interface

Security for Lightning Components: Lightning Locker

Control Flow Navigation from an Aura Component

By default, users navigate a flow by clicking standard buttons at the bottom of each screen. The `lightning:availableForFlowScreens` interface provides two attributes to help you fully customize your screen's navigation. To figure out which navigation actions are available for the screen, loop through the `availableActions` attribute. To programmatically trigger one of those actions, call the `navigateFlow` action from your JavaScript controller.

When you override the screen's navigation with an Aura component, remember to hide the footer so that the screen has only one navigation model.

IN THIS SECTION:

Flow Navigation Actions

The `availableActions` attribute lists the valid navigation actions for that screen.

[Customize the Flow Footer with an Aura Component]

To replace the flow footer with an Aura component, use the parameters that the `lightning:availableForFlowScreens` interface provides. The `availableActions` array lists which actions are available for the screen, and the `navigateFlow` action lets you invoke one of the available actions.

[Build a Custom Navigation Model for Your Flow Screens]

Since Aura components have access to a flow screen's navigation actions, you can fully customize how the user moves between screens. For example, hide the default navigation buttons and have the flow move to the next screen when the user selects a choice.

SEE ALSO:

Component Library: lightning:availableForFlowScreens Interface

Flow Navigation Actions

The `availableActions` attribute lists the valid navigation actions for that screen.

A screen's available actions are determined by:

- Where in the flow the screen is. For example, Previous isn't supported on the first screen in a flow, Finish is supported for only the last screen in a flow, and you can never have both Next and Finish.
- Whether the flow creator opted to hide any of the actions in the screen's Control Navigation settings. For example, if `Pause` is de-selected, the Pause action isn't included in availableActions.

Here are the possible actions, their default button label, and what's required for that action to be valid.

Action	Button Label	Description
`NEXT`	Next	Navigates to the next screen
`BACK`	Previous	Navigates to the previous screen
`PAUSE`	Pause	Saves the interview in its current state to the database, so that the user can resume it later
`RESUME`	Resume	Resumes a paused interview
`FINISH`	Finish	Finishes the interview. This action is available only before the final screen in the flow.

SEE ALSO:

Component Library: lightning:availableForFlowScreens Interface

Customize the Flow Footer with an Aura Component

To replace the flow footer with an Aura component, use the parameters that the `lightning:availableForFlowScreens` interface provides. The `availableActions` array lists which actions are available for the screen, and the `navigateFlow` action lets you invoke one of the available actions.

By default, the flow footer displays the available actions as standard buttons. Next and Finish use the brand variant style, and Previous and Pause use the neutral variant. Also, Pause floats left, while the rest of the buttons float right.

Example: This component (`c:flowFooter`) customizes the default flow footer in two ways.

- It swaps the Pause and Previous buttons, so that Previous floats to the left and Pause floats to the right with Next or Finish.
- It changes the label for the Finish button to Done.

`c:flowFooter` Component

Since the component implements `lightning:availableForFlowScreens`, it has access to the `availableActions` attribute, which contains the valid actions for the screen. The declared attributes, like `canPause` and `canBack`, determine which buttons to display. Those attributes are set by the JavaScript controller when the component initializes.

```
<aura:component access="global" implements="lightning:availableForFlowScreens">

   <!-- Determine which actions are available -->
   <aura:attribute name="canPause" type="Boolean" />
   <aura:attribute name="canBack" type="Boolean" />
   <aura:attribute name="canNext" type="Boolean" />
   <aura:attribute name="canFinish" type="Boolean" />
   <aura:handler name="init" value="{!this}" action="{!c.init}" />

   <div aura:id="actionButtonBar" class="slds-clearfix slds-p-top_medium">
      <!-- If Previous is available, display to the left -->
      <div class="slds-float_left">
         <aura:if isTrue="{!v.canBack}">
            <lightning:button aura:id="BACK" label="Previous"
               variant="neutral" onclick="{!c.onButtonPressed}" />
         </aura:if>
      </div>
      <div class="slds-float_right">
         <!-- If Pause, Next, or Finish are available, display to the right -->
         <aura:if isTrue="{!v.canPause}">
            <lightning:button aura:id="PAUSE" label="Pause"
               variant="neutral" onclick="{!c.onButtonPressed}" />
         </aura:if>
         <aura:if isTrue="{!v.canNext}">
            <lightning:button aura:id="NEXT" label="Next"
               variant="brand" onclick="{!c.onButtonPressed}" />
         </aura:if>
         <aura:if isTrue="{!v.canFinish}">
            <lightning:button aura:id="FINISH" label="Done"
               variant="brand" onclick="{!c.onButtonPressed}" />
         </aura:if>
      </div>
   </div>
</aura:component>
```

`c:flowFooter` Controller

The `init` function loops through the screen's available actions and determines which buttons the component should show. When the user clicks one of the buttons in the footer, the `onButtonPressed` function calls the `navigateFlow` action to perform that action.

```
({
   init : function(cmp, event, helper) {
      // Figure out which buttons to display
      var availableActions = cmp.get('v.availableActions');
      for (var i = 0; i < availableActions.length; i++) {
         if (availableActions[i] == "PAUSE") {
            cmp.set("v.canPause", true);
         } else if (availableActions[i] == "BACK") {
            cmp.set("v.canBack", true);
         } else if (availableActions[i] == "NEXT") {
            cmp.set("v.canNext", true);
         } else if (availableActions[i] == "FINISH") {
            cmp.set("v.canFinish", true);
         }
      }
   },

   onButtonPressed: function(cmp, event, helper) {
      // Figure out which action was called
      var actionClicked = event.getSource().getLocalId();
      // Fire that action
      var navigate = cmp.get('v.navigateFlow');
      navigate(actionClicked);
   }
})
```

Control Screen Navigation from a Child Component

If you're using a child component to handle the screen's navigation, pass the `availableActions` attribute down from the parent component – the one that implements `lightning:availableForFlowScreens`. You can pass the available actions by setting the child component's attributes, but you can't pass the action. Instead, use a custom event to send the selected action up to the parent component.

Example: `c:navigateFlow` Event

Create an event with an action attribute, so that you can pass the selected action into the event.

```
<aura:event type="APPLICATION" >
   <aura:attribute name="action" type="String"/>
</aura:event>
```

`c:flowFooter` Component

In your component, before the handler:

- Define an attribute to pass the screen's available actions from the parent component
- Register an event to pass the navigateFlow action to the parent component

```
<aura:attribute name="availableActions" type="String[]" />
<aura:registerEvent name="navigateFlowEvent" type="c:navigateFlow"/>
```

`c:flowFooter` Controller

Since `navigateFlow` is only available in the parent component, the `onButtonPressed` function fails. Update the `onButtonPressed` function so that it fires `navigateFlowEvent` instead.

```
onButtonPressed: function(cmp, event, helper) {
   // Figure out which action was called
   var actionClicked = event.getSource().getLocalId();
```

```
    // Call that action
    var navigate = cmp.getEvent("navigateFlowEvent");
    navigate.setParam("action", actionClicked);
    navigate.fire();
}
```

`c:flowParent` **Component**

In the parent component's markup, pass `availableActions` into the child component's `availableActions` attribute and the `handleNavigate` function into the child component's `navigateFlowEvent` event.

```
<c:flowFooter availableActions="{!v.availableActions}"
    navigateFlowEvent="{!c.handleNavigate}"/>
```

`c:flowParent` **Controller**

When `navigateFlowEvent` fires in the child component, the `handleNavigate` function calls the parent component's `navigateFlow` action, using the action selected in the child component.

```
handleNavigate: function(cmp, event) {
    var navigate = cmp.get("v.navigateFlow");
    navigate(event.getParam("action"));
}
```

SEE ALSO:

Customize the Flow Header with an Aura Component

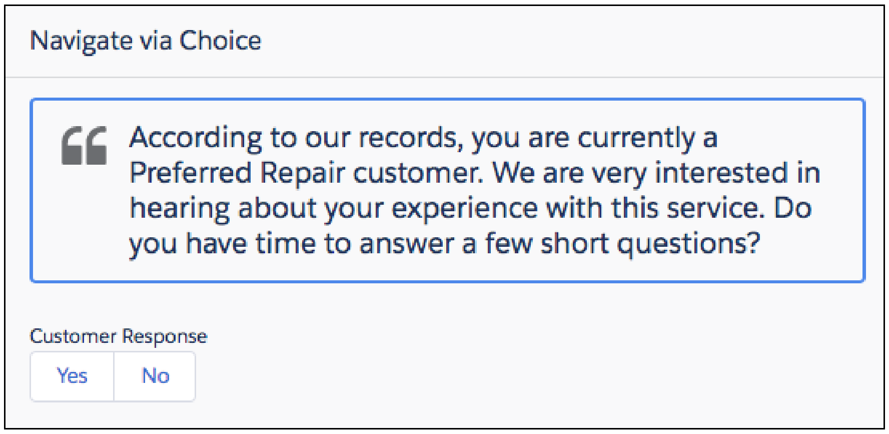
Dynamically Update a Flow Screen with an Aura Component

Component Library: lightning:availableForFlowScreens Interface

Build a Custom Navigation Model for Your Flow Screens

Since Aura components have access to a flow screen's navigation actions, you can fully customize how the user moves between screens. For example, hide the default navigation buttons and have the flow move to the next screen when the user selects a choice.

Example: This component (`c:choiceNavigation`) displays a script and a choice in the form of buttons.


Navigate via Choice

According to our records, you are currently a Preferred Repair customer. We are very interested in hearing about your experience with this service. Do you have time to answer a few short questions?

`c:choiceNavigation` Component

```
<aura:component implements="lightning:availableForFlowScreens" access="global" >
   <!-- Get the script text from the flow -->
   <aura:attribute name="scriptText" type="String" required="true" />
   <!-- Pass the value of the selected option back to the flow -->
   <aura:attribute name="value" type="String" />

   <!-- Display the script to guide the agent's call -->
   <div class="script-container">
     <div class="slds-card__header slds-grid slds-p-bottom_small slds-m-bottom_none">

         <div class="slds-media slds-media_center slds-has-flexi-truncate" >
            <div class="slds-media__figure slds-align-top">
               <h2><lightning:icon iconName="utility:quotation_marks"
                  title="Suggested script" /></h2>
            </div>
            <div class="slds-media__body">
               <ui:outputRichText class="script" value="{!v.scriptText}"/>
            </div>
         </div>
      </div>
   </div>
   <!-- Buttons for the agent to click, according to the customer’s response -->
   <div class="slds-p-top_large slds-p-bottom_large">
      <p><lightning:formattedText value="Customer Response"
         class="slds-text-body_small" /></p>
      <lightning:buttongroup >
         <lightning:button label="Yes" aura:id="Participate_Yes"
            variant="neutral" onclick="{!c.handleChange}"/>
         <lightning:button label="No" aura:id="Participate_No"
            variant="neutral" onclick="{!c.handleChange}"/>
     </lightning:buttongroup>
   </div>
</aura:component>
```

`c:choiceNavigation` Design

The design resource includes the `scriptText` attribute, so you can set the script from the flow.

```
<design:component>
   <design:attribute name="scriptText" label="Script Text"
      description="What the agent should say to the customer" />
</design:component>
```

`c:choiceNavigation` Style

```
.THIS.script-container {
   border: t(borderWidthThick) solid t(colorBorderBrand);
   border-radius: t(borderRadiusMedium);
}

.THIS .script {
   font-size: 1.125rem; /*t(fontSizeTextLarge)*/
   font-weight: t(fontWeightRegular);
   line-height: t(lineHeightHeading);
}
```

`c:choiceNavigation` Controller

When the user clicks either of the buttons, the JavaScript controller calls `navigateFlow("NEXT")`, which is the equivalent of the user clicking **Next**.

```
({
   handleChange : function(component, event, helper) {
      // When an option is selected, navigate to the next screen
      var response = event.getSource().getLocalId();
      component.set("v.value", response);
      var navigate = component.get("v.navigateFlow");
      navigate("NEXT");
   }
})
```

`defaultTokens.tokens`

The script in `c:choiceNavigation` uses tokens to stay in sync with the Salesforce Lightning Design System styles.

```
<aura:tokens extends="force:base" >
</aura:tokens>
```

SEE ALSO:

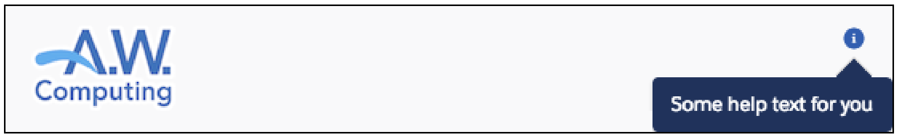
Component Library: lightning:availableForFlowScreens Interface

Customize the Flow Header with an Aura Component

To replace the flow header with an Aura component, use the `screenHelpText` parameter from the `lightning:availableForFlowScreens` interface.

By default, the flow header includes the title of the flow that's running and a button, where users can access screen-level help.

Example: Instead of displaying the flow title and the help button, this component (`c:flowHeader`) displays the company logo and the help button. The help text appears in a tooltip when the user hovers, instead of in a modal when the user clicks.



`c:flowHeader` Component

Since the component implements `lightning:availableForFlowScreens`, it has access to the `screenHelpText` attribute, which contains the screen's help text if it has any.

```
<aura:component access="global" implements="lightning:availableForFlowScreens">

   <div class="slds-p-top_medium slds-clearfix">
      <div class="slds-float_left">
         <!-- Display company logo -->
```

```
            <h2><img src="{!$Resource.Logo}" alt="A.W. Computing logo"/></h2>
         </div>
         <div class="slds-float_right" style="position:relative;">
            <aura:if isTrue="{!v.screenHelpText ne null}">
               <!-- If the screen has help text, display an info icon in the header.
                    On hover, display the screen's help text -->
               <lightning:helptext content="{!v.screenHelpText}" />
            </aura:if>
         </div>
      </div>
</aura:component>
```

SEE ALSO:

Customize the Flow Footer with an Aura Component

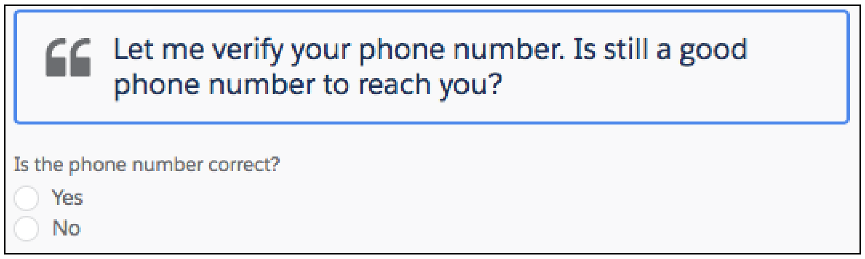
Dynamically Update a Flow Screen with an Aura Component

Component Library: lightning:availableForFlowScreens Interface

Dynamically Update a Flow Screen with an Aura Component

To conditionally display a field on your screen, build an Aura component that uses `aura:if` to check when parts of the component should appear.

Example: This component (`c:flowDynamicScreen`) displays a custom script component and a group of radio buttons. The component gets the contact's existing phone number from the flow, and uses that value to fill in the script.

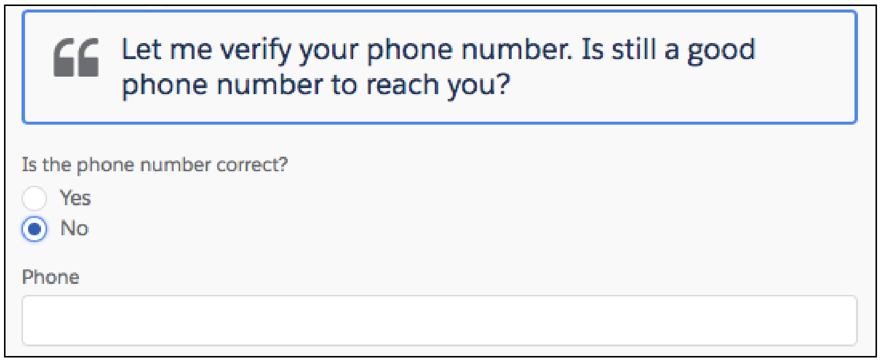

Let me verify your phone number. Is still a good phone number to reach you?

Is the phone number correct?

Yes

No

If the user selects the No radio button, the component displays an input, where the user can enter the new phone number.


Let me verify your phone number. Is still a good phone number to reach you?

Is the phone number correct?

Yes

[✓ No]

Phone

`c:flowDynamicScreen` **Component**

```
<aura:component access="global" implements="lightning:availableForFlowScreens">
   <aura:attribute name="oldPhone" type="String" />
   <aura:attribute name="newPhone" type="String" />
   <aura:attribute name="radioOptions" type="List" default="[
      {'label': 'Yes', 'value': 'false'},
      {'label': 'No', 'value': 'true'} ]"/>
   <aura:attribute name="radioValue" type="Boolean" />

   <!-- Displays script to guide the agent's call -->
   <div class="script-container">
     <div class="slds-card__header slds-grid slds-p-bottom_small slds-m-bottom_none">

         <div class="slds-media slds-media_center slds-has-flexi-truncate" >
            <div class="slds-media__figure slds-align-top">
               <h2><lightning:icon iconName="utility:quotation_marks"
                  title="Suggested script" /></h2>
            </div>
            <div class="slds-media__body">
               <!-- Inserts the user’s current number, pulled from the flow, into the
 script -->
               <ui:outputRichText class="script" value="{!'Let me verify your phone
number.
                  Is ' + v.oldPhone + ' still a good phone number to reach you?'}"/>
            </div>
         </div>
      </div>
   </div>
   <!-- Displays a radio button group to enter the customer’s response -->
   <div class="slds-p-top_medium slds-p-bottom_medium">
      <lightning:radioGroup aura:id="rbg_correct" name="rbg_correct"
         label="Is the phone number correct?"
         options="{! v.radioOptions }" value="{! v.radioValue }" />
      <!-- If the current number is wrong,
         displays a field to enter the correct number -->
      <aura:if isTrue="{!v.radioValue}">
         <lightning:input type="tel" aura:id="phone_updated" label="Phone"
            onblur="{!c.handleNewPhone}" class="slds-p-top_small"/>
      </aura:if>
   </div>
</aura:component>
```

`c:flowDynamicScreen` Style

```
.THIS.script-container {
   border: t(borderWidthThick) solid t(colorBorderBrand);
   border-radius: t(borderRadiusMedium);
}

.THIS .script {
   font-size: 1.125rem; /*t(fontSizeTextLarge)*/
   font-weight: t(fontWeightRegular);
   line-height: t(lineHeightHeading);
}
```

`c:flowDynamicScreen` **Controller**

When the user tabs out, or otherwise removes focus from the Phone input, the controller sets the `newPhone` attribute to the input value, so that you can reference the new number in the flow.

```
({
   handleNewPhone: function(cmp, event, helper) {
      cmp.set("v.newPhone", cmp.find('phone_updated').get('v.value'));
   }
})
```

`defaultTokens.tokens`

The script in `c:flowDynamicScreen` uses tokens to stay in sync with the Salesforce Lightning Design System styles.

```
<aura:tokens extends="force:base" >
</aura:tokens>
```

SEE ALSO:

Customize the Flow Header with an Aura Component

Customize the Flow Footer with an Aura Component

Component Library: lightning:availableForFlowScreens Interface

Create Flow Local Actions Using Aura Components

To execute client-side logic in your flow, build or modify custom Aura components to use as local actions in flows. For example, get data from third-party systems without going through the Salesforce server, or open a URL in another browser tab. Once you configure the Aura component's markup, client-side controller, and design resource, it's available in Flow Builder as a Core Action element.

Note:

- Lightning components in flows must comply with Lightning Locker restrictions.
- Flows that include Lightning components are supported only in Lightning runtime.
- Lightning components require a browser context to run, so flow action components are supported only in screen flows.

Example: Here's a sample "c:helloWorld" component and its client-side controller, which triggers a JavaScript alert that says `Hello, World`. In Flow Builder, local actions are available from the Core Action element.

```
<aura:component implements="lightning:availableForFlowActions" access="global">
    <aura:attribute name="greeting" type="String" default="Hello" access="global" />
    <aura:attribute name="subject" type="String" default="World" access="global" />
</aura:component>
```

```
({
   // When a flow executes this component, it calls the invoke method
   invoke : function(component, event, helper) {
      alert(component.get("v.greeting") + ", " + component.get("v.subject"));
   }
})
```

IN THIS SECTION:

SEE ALSO:

Component Library: lightning:availableForFlowActions Interface

Security for Lightning Components: Lightning Locker

Customize Flow Screens Using Aura Components

Configure the Component Markup and Design Resource for a Flow Action

Make your custom Aura components available as flow local actions by implementing the `lightning:availableForFlowActions` interface.

Tip: We recommend that you omit markup from local actions. Local actions tend to execute quickly, and any markup you add to them will likely disappear before the user can make sense of it. If you want to display something to users, check out Customize Flow Screens Using Aura Components instead.

Here's sample code for a simple "Hello World" component that sets a couple of attributes.

```
<aura:component implements="lightning:availableForFlowActions" access="global">
   <aura:attribute name="greeting" type="String" access="global" />
   <aura:attribute name="subject" type="String" access="global" />
</aura:component>
```

Note: Mark your resources, such as a component, with `access="global"` to make the resource usable outside of your own org. For example, you want a component to be usable in an installed package or by a Lightning App Builder user or an Experience Builder user in another org.

To make an attribute's value customizable in Flow Builder, add it to the component's design resource. That way, flow admins can pass values between that attribute and the flow when they configure the corresponding Core Action element.

With this sample design resource, flow admins can customize the values for the "Hello World" component's attributes.

```
<design:component>
   <design:attribute name="greeting" label="Greeting" />
   <design:attribute name="subject" label="Subject" />
</design:component>
```

A design resource describes the design-time behavior of a Lightning component—information that visual tools require to allow adding the component to a page or app. Adding this resource is similar to adding it for the Lightning App Builder.

When admins reference this component in a flow, they can pass data between the flow and the Aura component. Use the Set Input Values tab to set an attribute using values from the flow. Use the Store Output Values tab to store an attribute's value in a flow variable.

SEE ALSO:

Component Library: lightning:availableForFlowActions Interface

Configure the Client-Side Controller for a Flow Local Action

Configure the Client-Side Controller for a Flow Local Action

When a component is executed as a flow local action, the flow calls the `invoke` method in the client-side controller. To run the code asynchronously in your client-side controller, such as when you're making an XML HTTP request (XHR), return a Promise. When the method finishes or the Promise is fulfilled, control is returned back to the flow.

Asynchronous Code

When a Promise is resolved, the next element in the flow is executed. When a Promise is rejected or hits the timeout, the flow takes the local action's fault connector and sets `$Flow.FaultMessage` to the error message.

By default, the error message is "An error occurred when the elementName element tried to execute the c:myComponent component." To customize the error message in `$Flow.FaultMessage`, return it as a new Error object in the `reject()` call.

```
({
   invoke : function(component, event, helper) {
      return new Promise(function(resolve, reject) {
         // Do something asynchronously, like get data from
         // an on-premise database

         // Complete the call and return to the flow
         if (/* request was successful */) {
            // Set output values for the appropriate attributes
            resolve();
         } else {
            reject(new Error("My error message")); }
      });
   }
})
```

Note: If you're making callouts to an external server, add the external server to the allowlist in your org and enable or configure CORS in the external server.

Synchronous Code

When the method finishes, the next element in the flow is executed.

```
({
   invoke : function(component, event, helper) {
      // Do something synchronously, like open another browser tab
      // with a specified URL

      // Set output values for the appropriate attributes
```

```
    }
})
```

SEE ALSO:

Component Library: lightning:availableForFlowActions Interface

Cancel an Asynchronous Request in a Flow Local Action

Using External JavaScript Libraries

Cancel an Asynchronous Request in a Flow Local Action

If an asynchronous request times out, the flow executes the local action's fault connector and sets `$Flow.FaultMessage` to the error message. However, the original request isn't automatically canceled. To abort an asynchronous request, use the `cancelToken` parameter available in the `invoke` method.

Note: By default, requests time out after 120 seconds. To override the default, assign a different Integer to the component's `timeout` attribute.

Example: In this client-side controller, the `invoke` method returns a Promise. When the method has done all it needs to do, it completes the call and control returns to the flow.

- If the request is successful, the method uses `resolve()` to execute the next element in the flow after this action.
- If the request isn't successful, it uses `reject()` to execute the local action's fault connector and sets `$Flow.FaultMessage` to "My error message".
- If the request takes too long, it uses `cancelToken.promise.then` to abort the request.

```
({
   invoke : function(component, event, helper) {
      var cancelToken = event.getParam("arguments").cancelToken;

      return new Promise(function(resolve, reject) {
         var xhttp = new XMLHttpRequest();

         // Do something, like get data from
         // a database behind your firewall
         xhttp.onreadystatechange = $A.getCallback(function() {
            if (/* request was successful */) {
               // Complete the call and return to the flow
               resolve();
            } else {
               reject(new Error("My error message"));
            }
         });

         // If the Promise times out, abort the request and
         // pass set $Flow.FaultMessage to "Request timed out"
         cancelToken.promise.then(function(error) {
            xhttp.abort();
            reject(new Error("Request timed out."));
         });

      });
```

```
    }
})
```

SEE ALSO:

Component Library: lightning:availableForFlowActions Interface

Configure the Client-Side Controller for a Flow Local Action

Embed a Flow in a Custom Aura Component

Once you embed a flow in an Aura component, use JavaScript and Apex code to configure the flow at run time. For example, pass values into the flow or to control what happens when the flow finishes. `lightning:flow` supports only screen flows and autolaunched flows.

A *flow* is an application, built with Flow Builder, that collects, updates, edits, and creates Salesforce information.

To embed a flow in your Aura component, add the `<lightning:flow>` component to it.

```
<aura:component>
    <aura:handler name="init" value="{!this}" action="{!c.init}" />
    <lightning:flow aura:id="flowData" />
</aura:component>
```

```
({
    init : function (component) {
        // Find the component whose aura:id is "flowData"
        var flow = component.find("flowData");
        // In that component, start your flow. Reference the flow's API Name.
        flow.startFlow("myFlow");
    },
})
```

Note: When a page loads that includes a flow component, such as Lightning App Builder or an active Lightning page, the flow runs. Make sure that the flow doesn't perform any actions – such as create or delete records – before the first screen.

IN THIS SECTION:

Reference Flow Output Variable Values in a Wrapper Aura Component

When you embed a flow in an Aura component, you can display or reference the flow's variable values. Use the `onstatuschange` action to get values from the flow's output variables. Output variables are returned as an array.

Set Flow Input Variable Values from a Wrapper Aura Component

When you embed a flow in a custom Aura component, give the flow more context by initializing its variables. In the component's controller, create a list of maps, then pass that list to the startFlow method.

Control a Flow's Finish Behavior by Wrapping the Flow in a Custom Aura Component

By default, when a flow user clicks **Finish**, a new interview starts and the user sees the first screen of the flow again. By embedding a flow in a custom Aura component, you can shape what happens when the flow finishes by using the `onstatuschange` action. To redirect to another page, use one of the `force:navigateTo*` events such as `force:navigateToObjectHome` or `force:navigateToUrl`.

Resume a Flow Interview from an Aura Component

By default, users can resume interviews that they paused from the Paused Interviews component on their home page. To customize how and where users can resume their interviews, embed the `lightning:flow` component in a custom Aura component. In your client-side controller, pass the interview ID into the `resumeFlow` method.

SEE ALSO:

Component Library: lightning:flow Component

Reference Flow Output Variable Values in a Wrapper Aura Component

When you embed a flow in an Aura component, you can display or reference the flow's variable values. Use the `onstatuschange` action to get values from the flow's output variables. Output variables are returned as an array.

Note: The variable must allow output access. If you reference a variable that doesn't allow output access, attempts to get the variable are ignored.

Example: This example uses the JavaScript controller to pass the flow's accountName and numberOfEmployees variables into attributes on the component. Then, the component displays those values in output components.

```
<aura:component>
   <aura:attribute name="accountName" type="String" />
   <aura:attribute name="numberOfEmployees" type="Decimal" />

   <p><lightning:formattedText value="{!v.accountName}" /></p>
  <p><lightning:formattedNumber style="decimal" value="{!v.numberOfEmployees}" /></p>


   <aura:handler name="init" value="{!this}" action="{!c.init}"/>
   <lightning:flow aura:id="flowData" onstatuschange="{!c.handleStatusChange}" />
</aura:component>
```

```
({
   init : function (component) {
      // Find the component whose aura:id is "flowData"
      var flow = component.find("flowData");
      // In that component, start your flow. Reference the flow's API Name.
      flow.startFlow("myFlow");
   },

   handleStatusChange : function (component, event) {
      if(event.getParam("status") === "FINISHED") {
         // Get the output variables and iterate over them
         var outputVariables = event.getParam("outputVariables");
         var outputVar;
         for(var i = 0; i < outputVariables.length; i++) {
            outputVar = outputVariables[i];
            // Pass the values to the component's attributes
            if(outputVar.name === "accountName") {
               component.set("v.accountName", outputVar.value);
            } else {
               component.set("v.numberOfEmployees", outputVar.value);
            }
         }
```

```
        }
    },
})
```

SEE ALSO:

Component Library: lightning:flow Component

Set Flow Input Variable Values from a Wrapper Aura Component

When you embed a flow in a custom Aura component, give the flow more context by initializing its variables. In the component's controller, create a list of maps, then pass that list to the startFlow method.

Tip: We recommend using Lightning web components because they perform better and provide the latest functionality. See Embed a Flow in a Custom Lightning Web Component.

You can set variables only at the beginning of an interview, and the variables you set must allow input access. If you reference a variable that doesn't allow input access, attempts to set the variable are ignored.

For each variable you set, provide the variable's `name`, `type`, and `value`. For type, use the API name for the flow data type. For example, for a record variable use SObject, and for a text variable use String.

```
{
     name : "varName",
     type : "flowDataType",
     value : valueToSet
},
{
     name : "varName",
     type : "flowDataType",
     value : [ value1, value2]
}, ...
```

Example: This JavaScript controller sets values for a number variable, a date collection variable, and a couple of record variables. The Record data type in Flow Builder corresponds to SObject here.

```
({
   init : function (component) {
      // Find the component whose aura:id is "flowData"
      var flow = component.find("flowData");
      var inputVariables = [
         { name : "numVar", type : "Number", value: 30 },
         { name : "dateColl", type : "String", value: [ "2016-10-27", "2017-08-01" ]
},
         // Sets values for fields in the account record (sObject) variable. Id uses
         // the value of the component's accountId attribute. Rating uses a string.
         { name : "account", type : "SObject", value: {
             "Id" : component.get("v.accountId"),
             "Rating" : "Warm"
             }
          },
          // Set the contact record (sObject) variable to the value of the
          // component's contact attribute. We're assuming the attribute contains
          // the entire sObject for a contact record.
```

```
            { name : "contact", type : "SObject", value: component.get("v.contact") }
        ];
        flow.startFlow("myFlow", inputVariables);
    }
})
```

Example: Here's an example of a component that retrieves the most recently modified account via an Apex controller. The Apex controller passes the data to the flow's record variable through the JavaScript controller.

```
<aura:component controller="AccountController" >
   <aura:attribute name="account" type="Account" />
   <aura:handler name="init" value="{!this}" action="{!c.init}"/>
   <lightning:flow aura:id="flowData"/>
</aura:component>
```

```
public with sharing class AccountController {
    @AuraEnabled
    public static Account getAccount() {
       return [SELECT Id, Name, LastModifiedDate FROM Account
       ORDER BY LastModifiedDate DESC LIMIT 1];
    }
 }
```

```
({
    init : function (component) {
        // Create action to find an account
        var action = component.get("c.getAccount");

       // Add callback behavior for when response is received
        action.setCallback(this, function(response) {
            var state = response.getState();            if (state === "SUCCESS") {
             // Pass the account data into the component's account attribute
            component.set("v.account", response.getReturnValue());
             // Find the component whose aura:id is "flowData"
            var flow = component.find("flowData");
             // Set the account record (sObject) variable to the value of
             // the component's account attribute.
            var inputVariables = [
                {
                    name : "account",
                    type : "SObject",
                    value: component.get("v.account")
                }
            ];

             // In the component whose aura:id is flowData, start your flow
             // and initialize the account record (sObject) variable.
             // Reference the flow's API name.
            flow.startFlow("myFlow", inputVariables);
        }
            else {
                console.log("Failed to get account date.");
            }
    });
```

```
        // Send action to be executed
        $A.enqueueAction(action);
    }
})
```

SEE ALSO:

Component Library: lightning:flow Component

Which Custom Lightning Component Attribute Types Are Supported in Flows?

Control a Flow's Finish Behavior by Wrapping the Flow in a Custom Aura Component

By default, when a flow user clicks **Finish**, a new interview starts and the user sees the first screen of the flow again. By embedding a flow in a custom Aura component, you can shape what happens when the flow finishes by using the `onstatuschange` action. To redirect to another page, use one of the `force:navigateTo*` events such as `force:navigateToObjectHome` or `force:navigateToUrl`.

Tip: To control a flow's finish behavior at design time, make your custom Aura component available as a flow action by using the `lightning:availableForFlowActions` interface. To control what happens when an autolaunched flow finishes, check for the `FINISHED_SCREEN` status.

```
<aura:component access="global">
    <aura:handler name="init" value="{!this}" action="{!c.init}" />
    <lightning:flow aura:id="flowData" onstatuschange="{!c.handleStatusChange}" />
</aura:component>
```

```
// init function here
handleStatusChange : function (component, event) {
   if(event.getParam("status") === "FINISHED") {
        // Redirect to another page in Salesforce, or
        // Redirect to a page outside of Salesforce, or
        // Show a toast, or...
    }
}
```

Example: This function redirects the user to a case created in the flow by using the `force:navigateToSObject` event.

```
handleStatusChange : function (component, event) {
   if(event.getParam("status") === "FINISHED") {
      var outputVariables = event.getParam("outputVariables");
      var outputVar;
      for(var i = 0; i < outputVariables.length; i++) {
         outputVar = outputVariables[i];
         if(outputVar.name === "redirect") {
            var urlEvent = $A.get("e.force:navigateToSObject");
            urlEvent.setParams({
               "recordId": outputVar.value,
               "isredirect": "true"
            });
            urlEvent.fire();
         }
      }
```

```
    }
}
```

SEE ALSO:

Component Library: lightning:flow Component

Create Flow Local Actions Using Aura Components

Resume a Flow Interview from an Aura Component

By default, users can resume interviews that they paused from the Paused Interviews component on their home page. To customize how and where users can resume their interviews, embed the `lightning:flow` component in a custom Aura component. In your client-side controller, pass the interview ID into the `resumeFlow` method.

```
({
    init : function (component) {
        // Find the component whose aura:id is "flowData"
        var flow = component.find("flowData");

        // In that component, resume a paused interview. Provide the method with
        // the ID of the interview that you want to resume.
        flow.resumeFlow("pausedInterviewId");
     },
})
```

Example: This example shows how you can resume an interview—or start a new one. When users click **Survey Customer** from a contact record, the Aura component does one of two things.

- If the user has any paused interviews for the Survey Customers flow, it resumes the first one.
- If the user doesn't have any paused interviews for the Survey Customers flow, it starts a new one.

```
<aura:component controller="InterviewsController">
    <aura:handler name="init" value="{!this}" action="{!c.init}" />
    <lightning:flow aura:id="flowData" />
</aura:component>
```

This Apex controller gets a list of paused interviews by performing a SOQL query. If nothing is returned from the query, `getPausedId()` returns a null value, and the component starts a new interview. If at least one interview is returned from the query, the component resumes the first interview in that list.

```
public class InterviewsController {
   @AuraEnabled
   public static String getPausedId() {
      // Get the ID of the running user
      String currentUser = UserInfo.getUserId();
      // Find all of that user's paused interviews for the Survey customers flow
      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;
    }
}
```

If the Apex controller returned an interview ID, the client-side controller resumes that interview. If the Apex controller returned a null interview ID, the component starts a new interview.

```
({
   init : function (component) {
       //Create request for interview ID
       var action = component.get("c.getPausedId");
       action.setCallback(this, function(response) {
          var interviewId = response.getReturnValue();
          // Find the component whose aura:id is "flowData"
          var flow = component.find("flowData");
          // If an interview ID was returned, resume it in the component
          // whose aura:id is "flowData".
          if ( interviewId !== null ) {
             flow.resumeFlow(interviewID);
          }
          // Otherwise, start a new interview in that component. Reference
          // the flow's API Name.
          else {
             flow.startFlow("Survey_customers");
          }
       });
       //Send request to be enqueued
       $A.enqueueAction(action);
   },
})
```

SEE ALSO:

Component Library: lightning:flow Component

Display Flow Stages with an Aura Component

If you've added stages to your flow, display them to flow users with an Aura component, such as `lightning:progressindicator`.

To add a progress indicator component to your flow, you have two options:

- Wrap the progress indicator with a `lightning:flow` component in a parent component.

```
<aura:component>
   <lightning:progressindicator/>
   <lightning:flow/>
</aura:component>
```

- Add the progress indicator to your flow screen directly, by using a screen component.

IN THIS SECTION:

SEE ALSO:

Salesforce Help: Show Users Progress Through a Flow with Stages

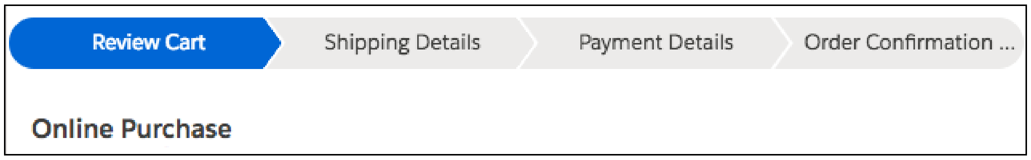
Display Flow Stages with a Progress Indicator on the Flow Screen

Display Flow Stages by Wrapping a Progress Indicator

If you're tracking stages in your flow, display them at runtime by creating a custom component that wraps a progress indicator with the `lightning:flow` component. Use the progress indicator to display the flow's active stages and current stage, and use the `lightning:flow` component to display the flow's screens. To pass the flow's active stages and current stage to the progress indicator, use the `lightning:flow` component's `onstatuschange` action.

Example: This `c:flowStages_global` component uses `lightning:progressindicator` to display the flow's stages and `lightning:flow` to display the flow.

Note: This example only applies to flows that have active stages.



`c:flowStages_global` Component

```
<aura:component implements="flexipage:availableForAllPageTypes" access="global" >
   <aura:attribute name="currentStage" type="Object"/>
   <aura:attribute name="activeStages" type="Object[]"/>
   <!-- Get flow name from the Lightning App Builder -->
   <aura:attribute name="flowName" type="String"/>

   <aura:handler name="init" value="{!this}" action="{!c.init}"/>
   <article class="slds-card">
      <lightning:progressIndicator aura:id="progressIndicator"
         currentStep="{!v.currentStage.name}" type="path"/>
        <lightning:flow aura:id="flow" onstatuschange="{!c.statusChange}"/>
   </article>
</aura:component>
```

`c:flowStages_global` Design

The design resource includes the `flowName` attribute, so you can specify which flow to start from Lightning App Builder.

```
<design:component>
   <design:attribute name="flowName" label="Flow Name"/>
</design:component>
```

`c:flowStages_global` Style

```
.THIS .slds-path__nav { margin-right: 0; }
.THIS .slds-path__item:only-child { border-radius: 15rem; }
```

`c:flowStages_global` Controller

The controller uses the `flowName` attribute to determine which flow to start.

Each time a new screen loads, the `onstatuschange` action fires, giving the controller access to a handful of parameters about the flow. The `currentStage` and `activeStages` parameters return the labels and names of the relevant stages.

When `onstatuschange` fires in this component, it calls the controller's `statusChange` method. That method passes the flow's `currentStage` and `activeStages` parameters into the component's attributes. For each item in the `activeStages` attribute, the method adds a `lightning:progressStep` component to the component markup.

```
({
   init : function(component, event, helper) {
      var flow = component.find("flow");
      flow.startFlow(component.get("v.flowName"));
   },

   // When each screen loads ...
   statusChange : function(component, event, helper) {
      // don't do anything if the flow doesn't have active stages
      if (!event.getParam("currentStage") || !event.getParam("activeStages")) {
          return;
      }
      // Pass $Flow.ActiveStages into the activeStages attribute
      // and $Flow.CurrentStage into the currentStage attribute
      component.set("v.currentStage", event.getParam("currentStage"));
      component.set("v.activeStages", event.getParam("activeStages"));

      var progressIndicator = component.find("progressIndicator");
      var body = [];

      for(let stage of component.get("v.activeStages")) {
         // For each stage in activeStages...
         $A.createComponent(
            "lightning:progressStep",
            {
               // Create a progress step where label is the
               // stage label and value is the stage name
               "aura:id": "step_" + stage.name,
               "label": stage.label,
               "value": stage.name
            },
            function(newProgressStep, status, errorMessage) {
               //Add the new step to the progress array
```

```
                    if (status === "SUCCESS") {
                    body.push(newProgressStep);
                    }
                    else if (status === "INCOMPLETE") {
                       // Show offline error
                       console.log("No response from server or client is offline.")
                    }
                    else if (status === "ERROR") {
                       // Show error message
                       console.log("Error: " + errorMessage);
                    }
                 }
              );
           }
           progressIndicator.set("v.body", body);
       }
})
```

SEE ALSO:

Salesforce Help: Show Users Progress Through a Flow with Stages

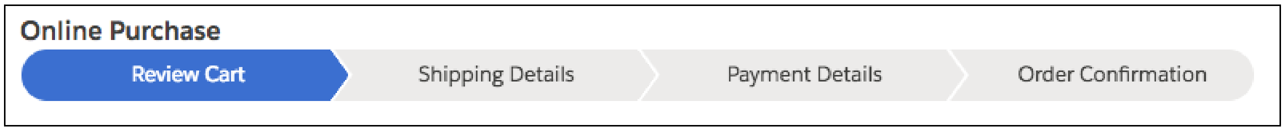
Display Flow Stages with an Aura Component

Aura Component Reference: Progress Indicator

Component Library: lightning:flow Component

Display Flow Stages with a Progress Indicator on the Flow Screen

If you track stages in your flow, display them at runtime by adding a custom component to the flow's screens. Create a progress indicator component that displays the flow's active stages and current stage, and make sure that it's available for flow screens. When you add the component to each flow screen, pass the `$Flow.ActiveStages` and `$Flow.CurrentStage` global variables into the component's attributes.



1. Create the custom `flowStages` component.

   The `flowStages` component uses `lightning:progressindicator` to display the flow's stages.

```
<aura:component implements="lightning:availableForFlowScreens">
    <!-- Attributes that store $Flow.ActiveStages and $Flow.CurrentStage -->
    <aura:attribute name="stages" type="String[]"/>
    <aura:attribute name="currentStage" type="String"/>
    <aura:handler name="init" value="{!this}" action="{!c.init}"/>
    <a href="#"/>
    <lightning:progressIndicator
        aura:id="progressIndicator"
        currentStep="{!v.currentStage}"
        type="path"/>
</aura:component>
```

2. Create the design resource for the `flowStages` component.

   The design resource includes the `stages` and `currentStage` attributes so that they're available in Flow Builder.

```
<design:component>
   <design:attribute name="stages" label="Stages" description="what stages are active"/>

   <design:attribute name="currentStage" label="Current Stage" description="the current
 stage"/>
</design:component>
```

3. Create the CSS style resource for the `flowStages` component.

```
.THIS .slds-path__nav { margin-right: 0; }
.THIS .slds-path__item:only-child { border-radius: 15rem; }
```

4. Create the client-side controller for the `flowStages` component.

   For each item in the `stages` attribute, the `init` method adds a `lightning:progressStep` component to the `flowStages` component markup.

```
({
   init : function(component, event, helper) {
      var progressIndicator = component.find('progressIndicator');
      for (let step of component.get('v.stages')) {
         $A.createComponent(
            "lightning:progressStep",
            {
               "aura:id": "step_" + step,
               "label": step,
               "value": step
             },
             function(newProgressStep, status, errorMessage){
                // Add the new step to the progress array
                if (status === "SUCCESS") {
                   var body = progressIndicator.get("v.body");
                   body.push(newProgressStep);
                   progressIndicator.set("v.body", body);
                 }
                 else if (status === "INCOMPLETE") {
                    // Show offline error
                    console.log("No response from server, or client is offline.")
                  }
                  else if (status === "ERROR") {
                     // Show error message
                     console.log("Error: " + errorMessage);
                  }
              }
           );
        }
    }
})
```

5. Create a flow in Flow Builder.

   a. From Setup, in the Quick Find box, enter *`Flows`*, and then select **Flows**. Then click **New Flow**.

   b. Select **Start From Scratch**, and then click **Next**.
   c. Select **Screen Flow** as the flow type, and then click **Create**.

6. Configure the stages in your flow.
   a. Click the Manager panel icon, and then click **New Resource**. Then select **Stage**.
   b. Enter a label and order, and then specify whether the stage is active by default.

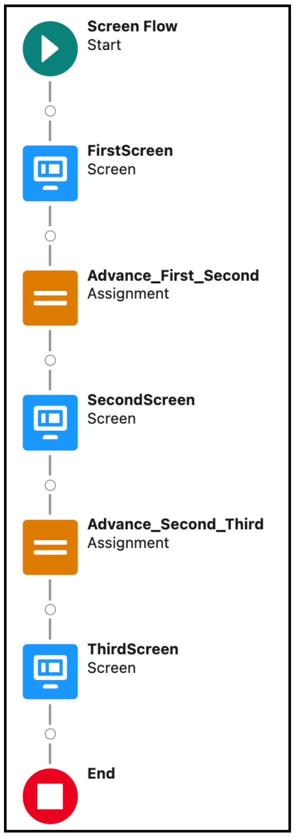
      If you select **Active by default**, the stage is added to `{!$Flow.ActiveStages}` when a flow interview starts.

7. Configure the screen elements in your flow.
   a. Click the Add Element icon on the canvas.
   b. Select the **Screen** interaction element.
   c. To the screen, add the custom **flowStages** component. For Current Stage, enter *`{!$Flow.CurrentStage}`*. For Stages, enter *`{!$Flow.ActiveStages}`*. In the Advanced section, select **Manually Assign Variables**.

8. Configure the assignment elements in your flow.
   a. Between each screen element, click the Add Element icon on the canvas.
   b. Select the **Assignment** logic element.
   c. Set the `Current Stage` variable equal to the following stage in the flow.

      For example, for the assignment element between the screens that contain the first and second stages, set the `Current Stage` equal to *`name_of_second_stage`*.

9. Save your flow.



SEE ALSO:

Salesforce Help: Show Users Progress Through a Flow with Stages

Display Flow Stages with an Aura Component

Display Flow Stages with an Aura Component

Component Library: lightning:availableForFlowScreens Interface

Add Components to Apps

When you're ready to add components to your app, first look at the built-in base components that Salesforce provides with the framework. You can also use these components by extending them or using composition to add them to custom components that you're building.

Note: For all the base components, see the Component Library on page 463. The `lightning` namespace includes many base components that implement visual elements common on web pages.

If you can't find a base component that meets your requirements, consider these options.

- Use design variations on page 121 on base components.
- Apply utility classes or custom CSS classes.
- Combine smaller base components into a more complex, custom component.
- Create your custom component from Lightning Design System blueprints.

Components are encapsulated and their internals stay private, while their public shape is visible to consumers of the component. This strong separation gives component authors freedom to change the internal implementation details and insulates component consumers from those changes.

The public shape of a component is defined by the attributes that can be set and the events that interact with the component. The shape is essentially the API for developers to interact with the component. To design a new component, think about the attributes that you want to expose and the events that the component can initiate or respond to.

After you've defined the shape of any new components, developers can work on the components in parallel. This approach is useful if you have a team working on an app.

To add a custom component to your app, see Using the Developer Console on page 4.

SEE ALSO:

- Component Composition
- Using Object-Oriented Development
- Component Attributes
- Communicating with Events

Integrate Your Custom Apps into the Chatter Publisher

Use the Chatter Rich Publisher Apps API to integrate your custom apps into the Chatter publisher. The Rich Publisher Apps API enables developers to attach any custom payload to a feed item. Rich Publisher Apps uses Lightning components for composition and rendering. We provide two Lightning interfaces and a Lightning event to assist with integration. You can package your apps and upload them to AppExchange. An Experience Builder site admin page provides a selector for choosing which five of your apps to add to the Chatter publisher for that site.

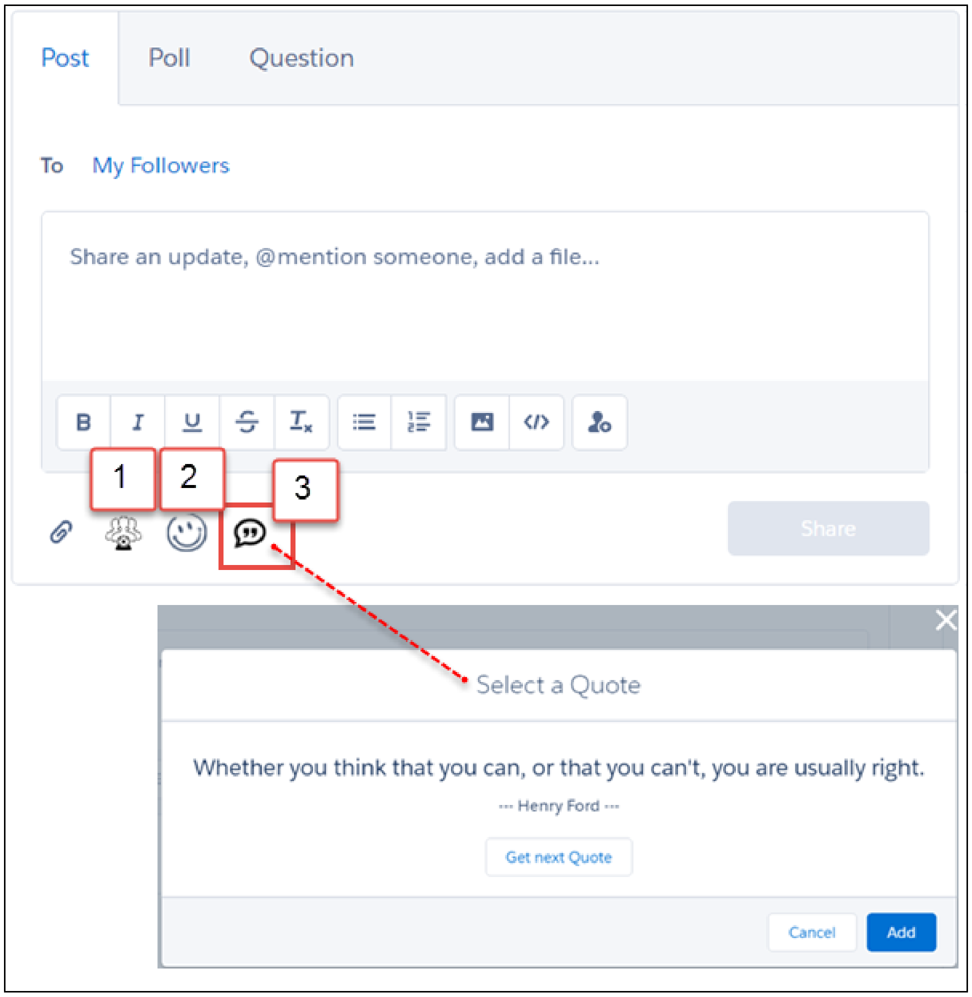
Note: Rich Publisher Apps are available to Experience Builder sites in topics, group, and profile feeds and in direct messages.

Use the `lightning:availableForChatterExtensionComposer` and `lightning:availableForChatterExtensionRenderer` interfaces with the `lightning:sendChatterExtensionPayload` event to integrate your custom apps into the Chatter publisher and carry your apps' payload into a Chatter feed.

Note: The payload must be an object.

Example: **Example of a Custom App Integrated into a Chatter Publisher**

This example shows a Chatter publisher with three custom app integrations. There are icons for a video meeting app (1), an emoji app (2), and an app for selecting a daily quotation (3).

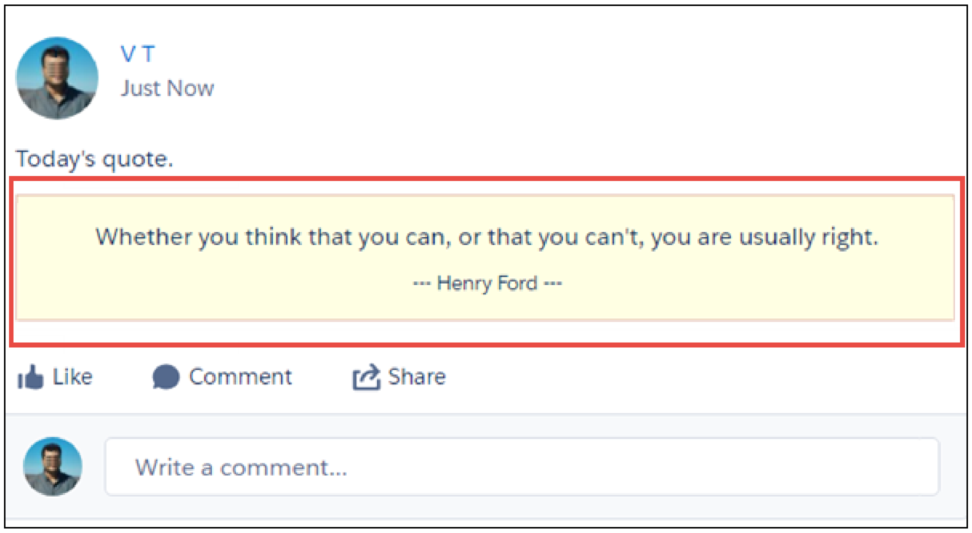


Whether you think that you can, or that you can't, you are usually right.

--- Henry Ford ---

Example: **Example of a Custom App Payload in a Chatter Feed Post**

This example shows the custom app's payload included in a Chatter feed.



Today's quote.

Whether you think that you can, or that you can't, you are usually right.

--- Henry Ford ---

The next sections describe how we integrated the custom quotation app with the Chatter publisher.

1. Set Up the Composer Component

For the composer component, we created component, controller, helper, and style files.

Here's the component markup in `quotesCompose.cmp`. In this file, we implement the `lightning:availableForChatterExtensionComposer` interface.

```
<aura:component implements="lightning:availableForChatterExtensionComposer">
    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <div class="container">
     <span class="quote" aura:id="quote"></span>
        <span class="author" aura:id="author"></span>
        <lightning:button label="Get next Quote" onclick="{!c.getQuote}"/>
    </div>

</aura:component>
```

Use your controller and helper to initialize the composer component and to get the quote from a source. When you get the quote, fire the event `sendChatterExtensionPayload`. Firing the event enables the **Add** button so the platform can associate the app's payload with the feed item. You can also add a title and description as metadata for the payload. The title and description are shown in a non-Lightning context, like Salesforce Classic.

```
getQuote: function(cmp, event, helper) {
    // get quote from the source
    var compEvent = cmp.getEvent("sendChatterExtensionPayload");
    compEvent.setParams({
        "payload" : "<payload object>",
        "extensionTitle" : "<title to use when extension is rendered>",
        "extensionDescription" : "<description to use when extension is rendered>"
    });
    compEvent.fire();
}
```

Add a CSS resource to your component bundle to style your composition component.

2. Set Up the Renderer Component

For the renderer component, we created component, controller, and style files.

Here's the component markup in `quotesRender.cmp`. In this file, we implement the `lightning:availableForChatterExtensionRenderer` interface, which provides the payload as an attribute in the component.

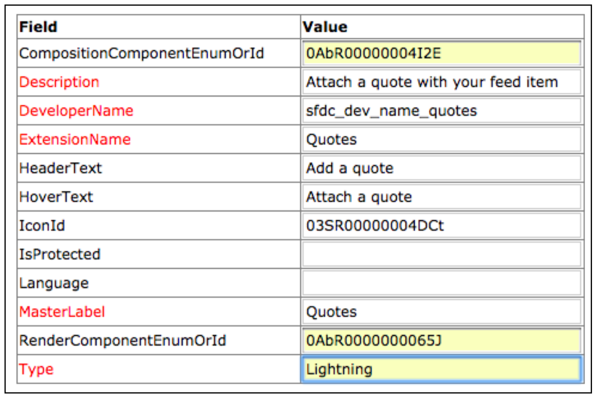
```
<aura:component implements="lightning:availableForChatterExtensionRenderer">
    <aura:attribute name="_quote" type="String"/>
    <aura:attribute name="_author" type="String"/>
    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <div class="container">
     <span class="quote" aura:id="quote">{!v._quote}</span>
        <span class="author" aura:id="author">--- {!v._author} ---</span>
    </div>
</aura:component>
```

You have a couple of ways of dealing with the payload. You can use the payload directly in the component `{!v.payload}`. You can use your controller to parse the payload provided by the `lightning:availableForChatterExtensionRenderer` interface and set its attributes yourself. Add a CSS resource to your renderer bundle to style your renderer component.

3. Set Up a New ChatterExtension Entity

After you create these components, open Postman or any tool that can make SOAP and REST API calls. Make sure that you're using at least API version 41.0. Log in to your org, and create a ChatterExtension entity using the Salesforce SOAP API.

Provide values for ChatterExtension fields (see ChatterExtension for values and descriptions).



Field	Value
CompositionComponentEnumOrId	0AbR00000004I2E
Description	Attach a quote with your feed item
DeveloperName	sfdc_dev_name_quotes
ExtensionName	Quotes
HeaderText	Add a quote
HoverText	Attach a quote
IconId	03SR00000004DCt
IsProtected	
Language	
MasterLabel	Quotes
RenderComponentEnumOrId	0AbR0000000065J
Type	Lightning

Get the `IconId` for the file asset. Go to Postman, or your preferred tool, and make a new POST request for creating a file asset with a `fileId` from your org. The filepath is `/services/data/v41.0/connect/files/<fileid>/asset`. Replace the version number with the current version.

Note: Rich Publisher Apps information is cached, so there can be a 5-minute wait before your app appears in the publisher.

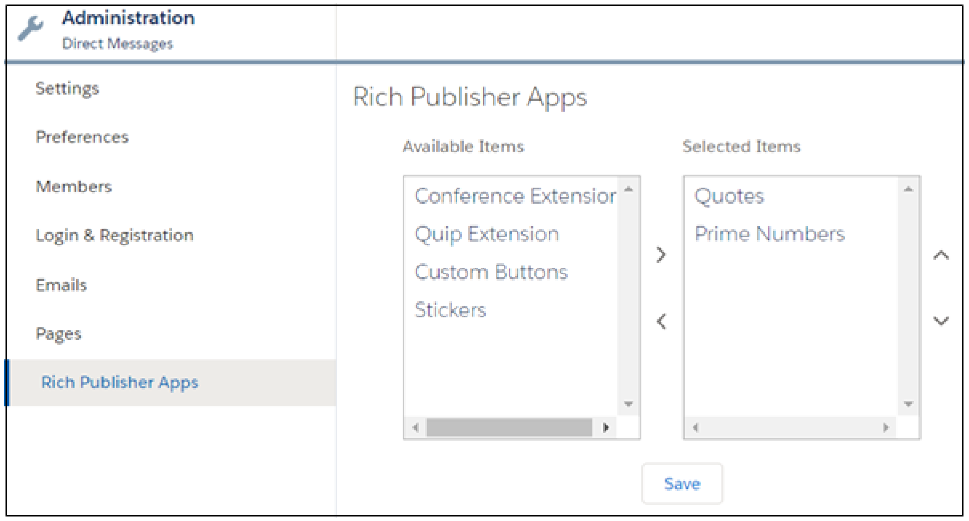
4. Package Your App and Upload It to AppExchange

The Second-Generation Managed Packaging Developer Guide provides useful information about packaging your apps and publishing them on AppExchange.

5. Select the Apps to Embed in the Chatter Publisher

An admin page is available in each Experience Builder site for selecting and arranging the apps to show in the Chatter publisher. Select up to five apps, and arrange them in the order you like. The order you set here controls the order the app icons appear in the publisher.

In your site, go to Experience Workspaces and open the Administration page. Click **Rich Publisher Apps** to open the page.



After you move apps to the Selected Items column and click **Save**, the selected apps appear in the Chatter Publisher.

Using Background Utility Items

Implement the `lightning:backgroundUtilityItem` interface to create a component that fires and responds to events without rendering in the utility bar.

Note: Lightning Web Components (LWC) doesn't currently support working with background utility items.

This component implements `lightning:backgroundUtilityItem` and listens for `lightning:tabCreated` events when the app loads. The component prevents more than 5 tabs from opening.

```
<aura:component implements="lightning:backgroundUtilityItem">
    <aura:attribute name="limit" default="5" type="Integer" />
    <aura:handler event="lightning:tabCreated" action="{!c.onTabCreated}" />
    <lightning:workspaceAPI aura:id="workspace" />
</aura:component>
```

When a tab is created, the event handler calls `onTabCreated` in the component's controller and checks how many tabs are open. If the number of tabs is more than 5, the leftmost tab automatically closes.

```
({
    onTabCreated: function(cmp) {
        var workspace = cmp.find("workspace");
        var limit = cmp.get("v.limit");
        workspace.getAllTabInfo().then(function (tabInfo) {
            if (tabInfo.length > limit) {
                workspace.closeTab({
                    tabId: tabInfo[0].tabId
                });
            }
        });
    }
})
```

Background utility items are added to an app the same way normal utility items are, but they don't appear in the utility bar. The icon appears next to background utility items on the utility item list. If you have only background utility items in your utility bar, the utility bar doesn't appear in your app. You need at least one non-background utility item in your utility bar for it to appear.

Use Lightning Components in Visualforce Pages

Add Aura components to your Visualforce pages to combine features that use both solutions. Implement new functionality using Aura components and then use it with existing Visualforce pages.

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*.

There are three steps to add Aura components to a Visualforce page.

1. Add the Lightning Components for Visualforce JavaScript library to your Visualforce page using the `<apex:includeLightning/>` component.
2. Create and reference a Lightning Out app that declares your component dependencies.
3. Write a JavaScript function that creates the component on the page using `$Lightning.createComponent()`.

Add the Lightning Components for Visualforce JavaScript Library

Add `<apex:includeLightning/>` at the beginning of your page. This component loads the JavaScript file used by Lightning Components for Visualforce.

Important: 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.

Create and Reference a Lightning Out App

To use Lightning Components for Visualforce, define component dependencies by referencing a Lightning Out app. This app is globally accessible and extends `ltng:outApp`. The app declares dependencies on any Lightning component that it uses.

Here's an example of a Lightning Out app named `lcvfTest.app`. The app uses the `<aura:dependency>` tag to indicate that it uses the standard Lightning component `lightning:button`.

```
<aura:application access="GLOBAL" extends="ltng:outApp">
    <aura:dependency resource="lightning:button"/>
</aura:application>
```

Note: Extending from `ltng:outApp` adds SLDS resources to the page so that your Lightning components can be styled with the Salesforce Lightning Design System (SLDS). If you don't want SLDS resources added to the page, extend from `ltng:outAppUnstyled` instead.

To reference this app on your page, use this JavaScript code, where *`theNamespace`* is the namespace prefix for the app. That is, either your org's namespace or the namespace of the managed package that provides the app.

```
$Lightning.use("theNamespace:lcvfTest", function() {});
```

If the app is defined in your org (that is, not in a managed package), you can use the default "c" namespace instead, as shown in the next example. If your org doesn't have a namespace defined, you *must* use the default namespace.

For details about creating a Lightning Out app, see Lightning Out Dependencies in the *Lightning Web Components Developer Guide*.

Creating a Component on a Page

Finally, add your top-level component to a page using `$Lightning.createComponent(String type, Object attributes, String domLocator, function callback)`. This function is similar to `$A.createComponent()`, but it includes an additional parameter, `domLocator`, that specifies the DOM element where you want the component inserted.

Let's look at a sample Visualforce page that creates a `lightning:button` using the `lcvfTest.app` from the previous example.

```
<apex:page>
    <apex:includeLightning />
    <div id="lightning" />
    <script>
        $Lightning.use("c:lcvfTest", function() {
            $Lightning.createComponent("lightning:button",
                { label : "Press Me!" },
                "lightning",
                function(cmp) {
                    console.log("button was created");
                    // do some stuff
                }
            );
        });
    </script>
</apex:page>
```

The `$Lightning.createComponent()` call creates a button with a "Press Me!" label. The button is inserted in a DOM element with the ID "lightning". After the button is added and active on the page, the callback function is invoked and executes a `console.log()` statement. The callback receives the component created as its only argument. In this simple example, the button isn't configured to do anything.

Important: You can call `$Lightning.use()` multiple times on a page, but all calls must reference the same Lightning dependency app.

For details about using `$Lightning.use()` and `$Lightning.createComponent()`, see Lightning Out Markup in the *Lightning Web Components Developer Guide*.

Limitations

If a Visualforce page contains an Aura component, you can't render the Visualforce page as a PDF.

Browser Third-Party Cookies

Lightning components set cookies in a user's browser. Because Lightning components and Visualforce are served from different domains, these cookies are "third-party" cookies.

You can use several approaches for enabling Lightning components in Visualforce to work with third-party cookies. See Enable Browser Third-Party Cookies for Lightning Out in the *Lightning Web Components Developer Guide*.

Use Aura and Lightning Web Components Outside of Salesforce with Lightning Out (Beta)

To run components outside of Salesforce servers, use Lightning Out, a special type of standalone Aura app. Whether it's a Node.js app running on Heroku or a department server inside the firewall, add your components as dependencies to a Lightning Out app. Then run the Lightning Out app wherever your users are.

Important: This feature 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.

Lightning Out supports both Aura components and Lightning web components. The setup process is the same for both component frameworks. We recommend using Lightning web components for the most modern, performant, and responsive functionality.

See Use Components Outside Salesforce with Lightning Out (Beta) in the *Lightning Web Components Developer Guide*.

SEE ALSO:

Use Lightning Web Components instead of Aura Components

Lightning Web Components Developer Guide: Use Components Outside Salesforce with Lightning Out (Beta)

Lightning Container

Upload an app developed with a third-party framework as a static resource, and host the content in an Aura component using `lightning:container`. Use `lightning:container` to use third-party frameworks like AngularJS or React within your Lightning pages.

The `lightning:container` component hosts content in an iframe. You can implement communication to and from the framed application, allowing it to interact with the Lightning component. `lightning:container` provides the `message()` method, which you can use in the JavaScript controller to send messages to the application. In the component, specify a method for handling messages with the `onmessage` attribute.

IN THIS SECTION:

Lightning Container Component Limits
Understand the limits of `lightning:container`.

lightning:container NPM Module Reference
Use methods included in the lightning:container NPM module in your JavaScript code to send and receive messages to and from your custom Aura component.

Using a Third-Party Framework

`lightning:container` allows you to use an app developed with a third-party framework, such as AngularJS or React, in an Aura component. Upload the app as a static resource.

Your application must have a launch page, which is specified with the `lightning:container src` attribute. By convention, the launch page is `index.html`, but you can specify another launch page by adding a manifest file to your static resource. The following example shows a simple Aura component that references `myApp`, an app uploaded as a static resource, with a launch page of `index.html`.

```
<aura:component>
    <lightning:container src="{!$Resource.myApp + '/index.html'}" />
</aura:component>
```

The contents of the static resource are up to you. It should include the JavaScript that makes up your app, any associated assets, and a launch page.

As in other Aura components, you can specify custom attributes. This example references the same static resource, `myApp`, and has three attributes, `messageToSend`, `messageReceived`, and `error`. Because this component includes `implements="flexipage:availableForAllPageTypes"`, it can be used in the Lightning App Builder and added to Lightning pages.

```
<aura:component access="global" implements="flexipage:availableForAllPageTypes" >
    <aura:attribute access="private" name="messageToSend" type="String" default=""/>
    <aura:attribute access="private" name="messageReceived" type="String" default=""/>
    <aura:attribute access="private" name="error" type="String" default=""/>

    <div>
        <lightning:input name="messageToSend" value="{!v.messageToSend}" label="Message
to send to React app: "/>
        <lightning:button label="Send" onclick="{!c.sendMessage}"/>
        <br/>
       <lightning:textarea value="{!v.messageReceived}" label="Message received from React
 app: "/>
        <br/>
        <aura:if isTrue="{! !empty(v.error)}">
            <lightning:textarea name="errorTextArea" value="{!v.error}" label="Error: "/>

        </aura:if>

        <lightning:container aura:id="ReactApp"
                             src="{!$Resource.SendReceiveMessages + '/index.html'}"
                             onmessage="{!c.handleMessage}"
                             onerror="{!c.handleError}"/>
    </div>
</aura:component>
```

The component includes a `lightning:input` element, allowing users to enter a value for `messageToSend`. When a user hits **Send**, the component calls the controller method `sendMessage`. This component also provides methods for handling messages and errors.

This snippet doesn't include the component's controller or other code, but don't worry. We'll dive in, break it down, and explain how to implement message and error handling as we go in Sending Messages from the Lightning Container Component and Handling Errors in Your Container.

SEE ALSO:

Lightning Container

Sending Messages from the Lightning Container Component

Handling Errors in Your Container

Sending Messages from the Lightning Container Component

Use the `onmessage` attribute of `lightning:container` to specify a method for handling messages to and from the contents of the component—that is, the embedded app. The contents of `lightning:container` are wrapped within an iframe, and this method allows you to communicate across the frame boundary.

This example shows an Aura component that includes `lightning:container` and has three attributes, `messageToSend`, `messageReceived`, and `error`.

This example uses the same code as the one in Using a Third-Party Framework.

```
<aura:component access="global" implements="flexipage:availableForAllPageTypes" >
    <aura:attribute access="private" name="messageToSend" type="String" default=""/>
    <aura:attribute access="private" name="messageReceived" type="String" default=""/>
    <aura:attribute access="private" name="error" type="String" default=""/>

    <div>
        <lightning:input name="messageToSend" value="{!v.messageToSend}" label="Message
to send to React app: "/>
        <lightning:button label="Send" onclick="{!c.sendMessage}"/>
        <br/>
       <lightning:textarea value="{!v.messageReceived}" label="Message received from React
 app: "/>
        <br/>
        <aura:if isTrue="{! !empty(v.error)}">
            <lightning:textarea name="errorTextArea" value="{!v.error}" label="Error: "/>

        </aura:if>

        <lightning:container aura:id="ReactApp"
                             src="{!$Resource.SendReceiveMessages + '/index.html'}"
                             onmessage="{!c.handleMessage}"
                             onerror="{!c.handleError}"/>
    </div>
</aura:component>
```

`messageToSend` represents a message sent from Salesforce to the framed app, while `messageReceived` represents a message sent by the app to the Aura component. `lightning:container` includes the required `src` attribute, an `aura:id`, and the `onmessage` attribute. The `onmessage` attribute specifies the message-handling method in your JavaScript controller, and the `aura:id` allows that method to reference the component.

This example shows the component's JavaScript controller.

```
({
    sendMessage : function(component, event, helper) {

        var msg = {
            name: "General",
            value: component.get("v.messageToSend")
        };
        component.find("ReactApp").message(msg);
    },

    handleMessage: function(component, message, helper) {
        var payload = message.getParams().payload;
        var name = payload.name;
        if (name === "General") {
            var value = payload.value;
            component.set("v.messageReceived", value);
        }
        else if (name === "Foo") {
            // A different response
        }
    },

    handleError: function(component, error, helper) {
        var e = error;
    }
})
```

This code does a couple of different things. The `sendMessage` action sends a message from the enclosing Aura component to the embedded app. It creates a variable, `msg`, that has a JSON definition including a `name` and a `value`. This definition of the message is user-defined—the message's payload can be a value, a structured JSON response, or something else. The `messageToSend` attribute of the Aura component populates the `value` of the message. The method then uses the component's `aura:id` and the `message()` function to send the message back to the Aura component.

The `handleMessage` method receives a message from the embedded app and handles it appropriately. It takes a component, a message, and a helper as arguments. The method uses conditional logic to parse the message. If this is the message with the `name` and `value` we're expecting, the method sets the Aura component's `messageReceived` attribute to the `value` of the message. Although this code only defines one message, the conditional statement allows you to handle different types of message, which are defined in the `sendMessage` method.

The handler code for sending and receiving messages can be complicated. It helps to understand the flow of a message between the Aura component, its controller, and the app. The process begins when user enters a message as the `messageToSend` attribute. When the user clicks **Send**, the component calls `sendMessage`. `sendMessage` defines the message payload and uses the `message()` method to send it to the app. Within the static resource that defines the app, the specified message handler function receives the message. Specify the message handling function within your JavaScript code using the lightning-container module's `addMessageHandler()` method. See the lightning:container NPM Module Reference for more information.

When `lightning:container` receives a message from the framed app, it calls the component controller's `handleMessage` method, as set in the `onmessage` attribute of `lightning:container`. The `handleMessage` method takes the message, and sets its value as the `messageReceived` attribute. Finally, the component displays `messageReceived` in a `lightning:textarea`.

This is a simple example of message handling across the container. Because you implement the controller-side code and the functionality of the app, you can use this functionality for any kind of communication between Salesforce and the app embedded in `lightning:container`.

Important: Don't send cryptographic secrets like an API key in a message. It's important to keep your API key secure.

SEE ALSO:

- Lightning Container
- Using a Third-Party Framework
- Handling Errors in Your Container

Sending Messages to the Lightning Container Component

Use the methods in the lightning-container NPM module to send messages from the JavaScript code framed by `lightning:container`.

The Lightning-container NPM module provides methods to send and receive messages between your JavaScript app and the Lightning container component. You can see the lightning-container module on the NPM website.

Add the lightning-container module as a dependency in your code to implement the messaging framework in your app.

```
import LCC from 'lightning-container';
```

`lightning-container` must also be listed as a dependency in your app's `package.json` file.

The code to send a message to `lightning:container` from the app is simple. This code corresponds to the code samples in Sending Messages from the Lightning Container Component and Handling Errors in Your Container.

```
sendMessage() {
  LCC.sendMessage({name: "General", value: this.state.messageToSend});
}
```

This code, part of the static resource, sends a message as an object containing a name and a value, which is user-defined.

When the app receives a message, it's handled by the function mounted by the `addMessageHandler()` method. In a React app, functions must be mounted to be part of the document-object model and rendered in the output.

The lightning-container module provides similar methods for defining a function to handle errors in the messaging framework. For more information, see lightning:container NPM Module Reference

Important: Don't send cryptographic secrets like an API key in a message. It's important to keep your API key secure.

Handling Errors in Your Container

Handle errors in Lightning container with a method in your component's controller.

This example uses the same code as the examples in Using a Third-Party Framework and Sending Messages from the Lightning Container Component.

In this component, the `onerror` attribute of `lightning:container` specifies `handleError` as the error handling method. To display the error, the component markup uses a conditional statement, and another attribute, `error`, for holding an error message.

```
<aura:component access="global" implements="flexipage:availableForAllPageTypes" >

    <aura:attribute access="private" name="messageToSend" type="String" default=""/>
    <aura:attribute access="private" name="messageReceived" type="String" default=""/>
    <aura:attribute access="private" name="error" type="String" default=""/>

    <div>
        <lightning:input name="messageToSend" value="{!v.messageToSend}" label="Message
to send to React app: "/><lightning:button label="Send" onclick="{!c.sendMessage}"/>

        <br/>

        <lightning:textarea name="messageReceived" value="{!v.messageReceived}"
label="Message received from React app: "/>

        <br/>
        <aura:if isTrue="{! !empty(v.error)}">
            <lightning:textarea name="errorMessage" value="{!v.error}" label="Error: "/>
        </aura:if>

        <lightning:container aura:id="ReactApp"
                             src="{!$Resource.SendReceiveMessages + '/index.html'}"
                             onmessage="{!c.handleMessage}"
                             onerror="{!c.handleError}"/>
    </div>

</aura:component>
```

This is the component's controller.

```
({
    sendMessage : function(component, event, helper) {

        var msg = {
            name: "General",
            value: component.get("v.messageToSend")
        };
        component.find("ReactApp").message(msg);
    },

    handleMessage: function(component, message, helper) {
        var payload = message.getParams().payload;
        var name = payload.name;
        if (name === "General") {
            var value = payload.value;
            component.set("v.messageReceived", value);
        }
        else if (name === "Foo") {
            // A different response
        }
    },

    handleError: function(component, error, helper) {
        var description = error.getParams().description;
        component.set("v.error", description);
    }
})
```

If the Lightning container application throws an error, the error handling function sets the `error` attribute. Then, in the component markup, the conditional expression checks if the error attribute is empty. If it isn't, the component populates a `lightning:textarea` element with the error message stored in `error`.

SEE ALSO:

Lightning Container

Using a Third-Party Framework

Sending Messages from the Lightning Container Component

Using Apex Services from Your Container

Use the `lightning-container` NPM module to call Apex methods from your Lightning container component.

To call Apex methods from `lightning:container`, you must set the CSP level to `low` in the `manifest.json` file. A CSP level of `low` allows the Lightning container component load resources from outside of the Lightning domain.

This is an Aura component that includes a Lightning container component that uses Apex services:

```
<aura:component access="global" implements="flexipage:availableForAllPageTypes">

    <aura:attribute access="private" name="error" type="String" default=""/>

    <div>
        <aura:if isTrue="{! !empty(v.error)}">
            <lightning:textarea name="errorTextArea" value="{!v.error}" label="Error: "/>

        </aura:if>

        <lightning:container aura:id="ReactApp"
                             src="/ApexController/index.html"
                             onerror="{!c.handleError}"/>
    </div>

</aura:component>
```

This is the component's controller:

```
({
    handleError: function(component, error, helper) {
        var description = error.getParams().description;
        component.set("v.error", description);
    }
})
```

There's not a lot going on in the component's JavaScript controller—the real action is in the JavaScript app, uploaded as a static resource, that the Lightning container references.

```
import React, { Component } from 'react';
import LCC from "lightning-container";
import logo from './logo.svg';
import './App.css';

class App extends Component {

  callApex() {
    LCC.callApex("lcc1.ApexController.getAccount",
                 this.state.name,
                 this.handleAccountQueryResponse,
                 {escape: true});
  }

  handleAccountQueryResponse(result, event) {
    if (event.status) {
      this.setState({account: result});
    }
    else if (event.type === "exception") {
      console.log(event.message + " : " + event.where);
    }
  }

  render() {
    var account = this.state.account;

    return (
      <div className="App">
        <div className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h2>Welcome to LCC</h2>
        </div>
        <p className="App-intro">
          Account Name: <input type="text" id="accountName" value={this.state.name}
onChange={e => this.onAccountNameChange(e)}/><br/>
         <input type="submit" value="Call Apex Controller" onClick={this.callApex}/><br/>

          Id: {account.Id}<br/>
          Phone: {account.Phone}<br/>
          Type: {account.Type}<br/>
          Number of Employees: {account.NumberOfEmployees}<br/>
        </p>
      </div>
    );
  }

  constructor(props) {
    super(props);
    this.state = {
      name: "",
      account: {}
    };

    this.handleAccountQueryResponse = this.handleAccountQueryResponse.bind(this);
    this.onAccountNameChange = this.onAccountNameChange.bind(this);
    this.callApex = this.callApex.bind(this);
  }

  onAccountNameChange(e) {
    this.setState({name: e.target.value});
  }
}

export default App;
```

The first function, `callApex()`, uses the `LCC.callApex` method to call `getAccount`, an Apex method that gets and displays an account's information.

Lightning Container Component Limits

Understand the limits of `lightning:container`.

`lightning:container` has known limitations. You might observe performance and scrolling issues associated with the use of iframes. This component isn't designed for the multi-page model, and it doesn't integrate with browser navigation history.

If you navigate away from the page and a `lightning:container` component is on, the component doesn't automatically remember its state. The content within the iframe doesn't use the same offline and caching schemes as the rest of Lightning Experience.

Creating a Lightning app that loads a Lightning container static resource from another namespace is not supported. If you install a package, your apps should use the custom Lightning components published by that package, not their static resources directly. Any static resource you use as the `lightning:container src` attribute should have your own namespace.

Previous versions of `lightning:container` allowed developers to specify the Content Security Policy (CSP) of the iframed content. We removed this functionality for security reasons. The CSP level of all pages is now set to the highest level to provide the greatest security. Content can only be loaded from secure, approved domains. When `lightning:container` is used in Experience Cloud, the CSP setting in that Experience Builder site will be respected.

Apps that use `lightning:container` should work with data, not metadata. Don't use the session key for your app to manage custom objects or fields. You can use the session key to create and update object records.

Content in `lightning:container` is served from the Lightning container domain and is available in Lightning Experience, Experience Builder sites, and the Salesforce mobile app. `lightning:container` can't be used in Lightning pages that aren't served from the Lightning domain, such as Visualforce pages or in external apps through Lightning Out.

Important: You can't access the Salesforce REST API from the app inside of `lightning:container`. See the Spring '18 Release Notes for details.

IN THIS SECTION:

Lightning Container Component Security Requirements
Ensure that your Lightning container components meet security requirements.

SEE ALSO:

Lightning Container
Salesforce Help: Content Security Policy in Experience Builder Sites

Lightning Container Component Security Requirements

Ensure that your Lightning container components meet security requirements.

Namespace Validity

The Lightning container component's security measures check the validity of its namespaces. Suppose that you develop a `<lightning:container>` component with the namespace "vendor1." The static resource's namespace must also be "vendor1." If they don't match, an error message appears.

```
<aura:component>
  <lightning:container
    src="{!$Resource.vendor1__resource + '/code_belonging_to_vendor1'}"
    onmessage="{!c.vendor1__handles}"/>
<aura:component>
```

Static Resource Content Access

You can't use raw `<iframe>` elements to access a Lightning container component. The `<lightning:container>` component enforces this requirement with the query parameter `_CONFIRMATIONTOKEN`, which generates a unique ID for each user session. The following code isn't permitted, because the `<iframe>` src attribute doesn't contain a `_CONFIRMATIONTOKEN` query parameter.

```
<aura:component>
  <iframe
src="https://domain--vendor2.container.lightning.com/lcc/123456/vendor2__resource/index.html"/>
</aura:component>
```

Instead, use the `$Resource` global value provider to build the resource URL for the `<lightning:container>` component.

```
<aura:component>
  <lightning:container
    src="{!$Resource.vendor2__resource + '/index.html' }"/>
</aura:component>
```

Distribution Requirements

To upload a package to AppExchange, you must supply all the Lightning container component's original sources and dependencies. When you provide minified or transpiled code, you must also include the source files for that code and the source map (.js.map) files for the minified code.

`lightning:container` NPM Module Reference

Use methods included in the lightning:container NPM module in your JavaScript code to send and receive messages to and from your custom Aura component.

IN THIS SECTION:

`addErrorHandler()`

Mounts an error handling function, to be called when the messaging framework encounters an error.

Sample

Used within a JavaScript app uploaded as a static resource and referenced by `lightning:container`, this example mounts a message error handling function. In a React app, functions must be mounted to be part of the document-object model and rendered in the output.

```
componentDidMount() {
  LCC.addErrorHandler(this.onMessageError);
}
```

Arguments

Name	Type	Description
`handler: (errorMsg: string) => void)`	function	The function that handles error messages encountered in the messaging framework.

Response

None.

`addMessageHandler()`

Mounts a message handling function, used to handle messages sent from the Aura component to the framed JavaScript app.

Sample

Used within a JavaScript app uploaded as a static resource and referenced by `lightning:container`, this example mounts a message handling function. In a React app, functions must be mounted to be part of the document-object model and rendered in the output.

```
componentDidMount() {
    LCC.addMessageHandler(this.onMessage);
}

onMessage(msg) {
  let name = msg.name;
  if (name === "General") {
    let value = msg.value;
    this.setState({messageReceived: value});
  }
  else if (name === "Foo") {
    // A different response
  }
}
```

Arguments

Name	Type	Description
`handler: (userMsg: any) => void`	function	The function that handles messages sent from the Aura component.

Response

None.

`callApex()`

Makes an Apex call.

Sample

Used within a JavaScript app uploaded as a static resource and referenced by `lightning:container`, this example calls the Apex method `getAccount`.

```
callApex() {
  LCC.callApex("lcc1.ApexController.getAccount",
               this.state.name,
               this.handleAccountQueryResponse,
               {escape: true});
}
```

Arguments

Name	Type	Description
`fullyQualifiedApexMethodName`	string	The name of the Apex method.
`apexMethodParameters`	array	A JSON array of arguments for the Apex method.
`callbackFunction`	function	A callback function.
`apexCallConfiguration`	array	Configuration parameters for the Apex call.

Response

None.

`removeErrorHandler()`

Unmounts the error handling function.

When using React, it's necessary to unmount functions to remove them from the DOM and perform necessary cleanup.

Sample

Used within a JavaScript app uploaded as a static resource and referenced by `lightning:container`, this example unmounts a message error handling function. In a React app, functions must be mounted to be part of the document-object model and rendered in the output.

```
componentWillUnmount() {
  LCC.removeErrorHandler(this.onMessageError);
}
```

Arguments

Name	Type	Description
`handler: (errorMsg: string) => void)`	function	The function that handles error messages encountered in the messaging framework.

Response

None.

`removeMessageHandler()`

Unmounts the message-handling function.

When using React, it's necessary to unmount functions to remove them from the DOM and perform necessary cleanup.

Sample

Used within a JavaScript app uploaded as a static resource and referenced by `lightning:container`, this example unmounts a message handling function.

```
componentWillUnmount() {
  LCC.removeMessageHandler(this.onMessage);
}
```

Arguments

Name	Type	Description
`handler: (userMsg: any) => void`	function	The function that handles messages sent from the Aura component.

Response

None.

`sendMessage()`

Sends a message from the framed JavaScript code to the Aura component.

Sample

Used within a JavaScript app uploaded as a static resource and referenced by `lightning:container`, this example sends a message from the app to `lightning:container`.

```
sendMessage() {
  LCC.sendMessage({name: "General", value: this.state.messageToSend});
}
```

Arguments

Name	Type	Description
`userMsg`	any	While the data sent in the message is entirely under your control, by convention it's an object with name and value fields.

Response

None.

CHAPTER 5 Communicating with Events

In this chapter ...

The framework uses event-driven programming. You write handlers that respond to interface events as they occur. The events may or may not have been triggered by user interaction.

In the Aura Components programming model, events are fired from JavaScript controller actions. Events can contain attributes that can be set before the event is fired and read when the event is handled.

Events are declared by the `aura:event` tag in a `.evt` resource, and they can have one of two types: component or application.

Component Events

A component event is fired from an instance of a component. A component event can be handled by the component that fired the event or by a component in the containment hierarchy that receives the event.

Application Events

Application events follow a traditional publish-subscribe model. An application event is fired from an instance of a component. All components that provide a handler for the event are notified.

Note: Always try to use a component event instead of an application event, if possible. Component events can only be handled by components above them in the containment hierarchy so their usage is more localized to the components that need to know about them. Application events are best used for something that should be handled at the application level, such as navigating to a specific record. Application events allow communication between components that are in separate parts of the application and have no direct containment relationship.

Actions and Events

The framework uses events to communicate data between components. Events are usually triggered by a user action.

Actions

User interaction with an element on a component or app. User actions trigger events, but events aren't always explicitly triggered by user actions. This type of action is *not* the same as a client-side JavaScript controller, which is sometimes known as a *controller action*. The following button is wired up to a browser `onclick` event in response to a button click.

```
<lightning:button label = "Click Me" onclick = "{!c.handleClick}" />
```

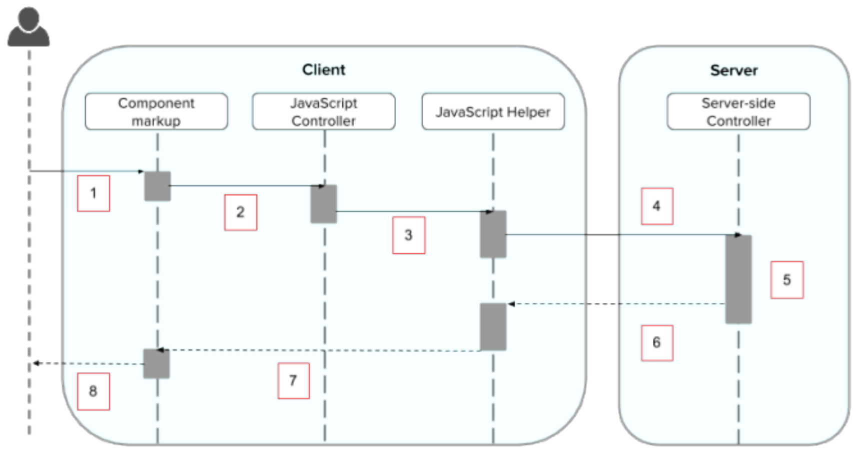
Clicking the button invokes the `handleClick` method in the component's client-side controller.

Events

A notification by the browser regarding an action. Browser events are handled by client-side JavaScript controllers, as shown in the previous example. A browser event is not the same as a framework *component event* or *application event*, which you can create and fire in a JavaScript controller to communicate data between components. For example, you can wire up the click event of a checkbox to a client-side controller, which fires a component event to communicate relevant data to a parent component.

Another type of event, known as a *system event*, is fired automatically by the framework during its lifecycle, such as during component initialization, change of an attribute value, and rendering. Components can handle a system event by registering the event in the component markup.

The following diagram describes what happens when a user clicks a button that requires the component to retrieve data from the server.



1. User clicks a button or interacts with a component, triggering a browser event. For example, you want to save data from the server when the button is clicked.
2. The button click invokes a client-side JavaScript controller, which provides some custom logic before invoking a helper function.
3. The JavaScript controller invokes a helper function. A helper function improves code reuse but it's optional for this example.
4. The helper function calls an Apex controller method and queues the action.
5. The Apex method is invoked and data is returned.
6. A JavaScript callback function is invoked when the Apex method completes.
7. The JavaScript callback function evaluates logic and updates the component's UI.

8. User sees the updated component.

SEE ALSO:

Handling Events with Client-Side Controllers
Detecting Data Changes with Change Handlers
Calling a Server-Side Action
Events Fired During the Rendering Lifecycle

Handling Events with Client-Side Controllers

A client-side controller handles events within a component. It's a JavaScript resource that defines the functions for all of the component's actions.

A client-side controller is a JavaScript object in object-literal notation containing a map of name-value pairs. Each name corresponds to a client-side action. Its value is the function code associated with the action. Client-side controllers are surrounded by parentheses and curly braces. Separate action handlers with commas (as you would with any JavaScript map).

```
({
    myAction : function(cmp, event, helper) {
        // add code for the action
    },

    anotherAction : function(cmp, event, helper) {
        // add code for the action
    }
})
```

Each action function takes in three parameters:

1. `cmp`—The component to which the controller belongs.
2. `event`—The event that the action is handling.
3. `helper`—The component's helper, which is optional. A helper contains functions that can be reused by any JavaScript code in the component bundle.

Creating a Client-Side Controller

A client-side controller is part of the component bundle. It is auto-wired via the naming convention, `*componentName*Controller.js`.

To create a client-side controller using the Developer Console, click **CONTROLLER** in the sidebar of the component.

Calling Client-Side Controller Actions

The following example component creates two buttons to contrast an HTML button with `<lightning:button>`, which is a standard Lightning component. Clicking on these buttons updates the `text` component attribute with the specified values. `target.get("v.label")` refers to the `label` attribute value on the button.

Component source

```
<aura:component>
    <aura:attribute name="text" type="String" default="Just a string. Waiting for change."/>

    <input type="button" value="Flawed HTML Button"
        onclick="alert('this will not work')"/>
    <br/>
    <lightning:button label="Framework Button" onclick="{!c.handleClick}"/>
    <br/>
    {!v.text}
</aura:component>
```

If you know some JavaScript, you might be tempted to write something like the first "Flawed" button because you know that HTML tags are first-class citizens in the framework. However, the "Flawed" button won't work because arbitrary JavaScript, such as the `alert()` call, in the component is ignored.

The framework has its own event system. DOM events are mapped to Lightning events, since HTML tags are mapped to Lightning components.

Any browser DOM element event starting with `on`, such as `onclick` or `onkeypress`, can be wired to a controller action. You can only wire browser events to controller actions.

The "Framework" button wires the `onclick` attribute in the `<lightning:button>` component to the `handleClick` action in the controller.

Client-side controller source

```
({
    handleClick : function(cmp, event) {
        var attributeValue = cmp.get("v.text");
        console.log("current text: " + attributeValue);

        var target = event.getSource();
        cmp.set("v.text", target.get("v.label"));
    }
})
```

The `handleClick` action uses `event.getSource()` to get the source component that fired this component event. In this case, the source component is the `<lightning:button>` in the markup.

The code then sets the value of the `text` component attribute to the value of the button's `label` attribute. The `text` component attribute is defined in the `<aura:attribute>` tag in the markup.

Tip: Use unique names for client-side and server-side actions in a component. A JavaScript function (client-side action) with the same name as an Apex method (server-side action) can lead to hard-to-debug issues. In debug mode, the framework logs a browser console warning about the clashing client-side and server-side action names.

Handling Framework Events

Handle framework events using actions in client-side component controllers. Framework events for common mouse and keyboard interactions are available with out-of-the-box components.

Accessing Component Attributes

In the `handleClick` function, notice that the first argument to every action is the component to which the controller belongs. One of the most common things you'll want to do with this component is look at and change its attribute values.

`cmp.get("v.attributeName")` returns the value of the ***`attributeName`*** attribute.

`cmp.set("v.attributeName", "attribute value")` sets the value of the ***`attributeName`*** attribute.

Invoking Another Action in the Controller

To call an action method from another method, put the common code in a helper function and invoke it using `helper.someFunction(cmp)`.

SEE ALSO:

- Sharing JavaScript Code in a Component Bundle
- Event Handling Lifecycle
- Creating Server-Side Logic with Controllers

Component Events

A component event is fired from an instance of a component. A component event can be handled by the component that fired the event or by a component in the containment hierarchy that receives the event.

IN THIS SECTION:

Component Event Propagation

The framework supports *capture* and *bubble* phases for the propagation of component events. These phases are similar to DOM handling patterns and provide an opportunity for interested components to interact with an event and potentially control the behavior for subsequent handlers.

Create Custom Component Events

Create a custom component event using the `<aura:event>` tag in a `.evt` resource. Events can contain attributes that can be set before the event is fired and read when the event is handled.

Fire Component Events

Fire a component event to communicate data to another component. A component event can be handled by the component that fired the event or by a component in the containment hierarchy that receives the event.

Handling Component Events

A component event can be handled by the component that fired the event or by a component in the containment hierarchy that receives the event.

SEE ALSO:

aura:method

Application Events

Handling Events with Client-Side Controllers

Advanced Events Example

What is Inherited?

Component Event Propagation

The framework supports *capture* and *bubble* phases for the propagation of component events. These phases are similar to DOM handling patterns and provide an opportunity for interested components to interact with an event and potentially control the behavior for subsequent handlers.

The component that fires an event is known as the source component. The framework allows you to handle the event in different phases. These phases give you flexibility for how to best process the event for your application.

The phases are:

Capture

The event is captured and trickles down from the application root to the source component. The event can be handled by a component in the containment hierarchy that receives the captured event.

Event handlers are invoked in order from the application root down to the source component that fired the event.

Any registered handler in this phase can stop the event from propagating, at which point no more handlers are called in this phase or the bubble phase.

Bubble

The component that fired the event can handle it. The event then bubbles up from the source component to the application root. The event can be handled by a component in the containment hierarchy that receives the bubbled event.

Event handlers are invoked in order from the source component that fired the event up to the application root.

Any registered handler in this phase can stop the event from propagating, at which point no more handlers are called in this phase.

Here's the sequence of component event propagation.

1. **Event fired**—A component event is fired.
2. **Capture phase**—The framework executes the capture phase from the application root to the source component until all components are traversed. Any handling event can stop propagation by calling `stopPropagation()` on the event.
3. **Bubble phase**—The framework executes the bubble phase from the source component to the application root until all components are traversed or `stopPropagation()` is called.

SEE ALSO:

Salesforce Developers Blog: An In-Depth Look at Lightning Component Events

Create Custom Component Events

Create a custom component event using the `<aura:event>` tag in a `.evt` resource. Events can contain attributes that can be set before the event is fired and read when the event is handled.

Use `type="COMPONENT"` in the `<aura:event>` tag for a component event. For example, this `c:compEvent` component event has one attribute with a name of `message`.

```
<!--c:compEvent-->
<aura:event type="COMPONENT">
    <!-- Add aura:attribute tags to define event shape.
         One sample attribute here. -->
    <aura:attribute name="message" type="String"/>
</aura:event>
```

The component that fires an event can set the event's data. To set the attribute values, call `event.setParam()` or `event.setParams()`. A parameter name set in the event must match the `name` attribute of an `<aura:attribute>` in the event. For example, if you fire `c:compEvent`, you could use:

```
event.setParam("message", "event message here");
```

The component that handles an event can retrieve the event data. To retrieve the attribute value in this event, call `event.getParam("message")` in the handler's client-side controller.

Fire Component Events

Fire a component event to communicate data to another component. A component event can be handled by the component that fired the event or by a component in the containment hierarchy that receives the event.

Register an Event

A component registers that it may fire an event by using `<aura:registerEvent>` in its markup. For example:

```
<aura:registerEvent name="sampleComponentEvent" type="c:compEvent"/>
```

We'll see how the value of the `name` attribute is used for firing and handling events.

Fire an Event

To get a reference to a component event in JavaScript, use `cmp.getEvent("evtName")` where `evtName` matches the `name` attribute in `<aura:registerEvent>`.

Use `fire()` to fire the event from an instance of a component. For example, in an action function in a client-side controller:

```
var compEvent = cmp.getEvent("sampleComponentEvent");
// Optional: set some data for the event (also known as event shape)
// A parameter's name must match the name attribute
// of one of the event's <aura:attribute> tags
// compEvent.setParams({"myParam" : myValue });
compEvent.fire();
```

SEE ALSO:

Fire Application Events

Handling Component Events

A component event can be handled by the component that fired the event or by a component in the containment hierarchy that receives the event.

Use `<aura:handler>` in the markup of the handler component. For example:

```
<aura:handler name="sampleComponentEvent" event="c:compEvent"
    action="{!c.handleComponentEvent}"/>
```

The `name` attribute in `<aura:handler>` must match the `name` attribute in the `<aura:registerEvent>` tag in the component that fires the event.

The `action` attribute of `<aura:handler>` sets the client-side controller action to handle the event.

The `event` attribute specifies the event being handled. The format is ***`namespace:eventName`***.

In this example, when the event is fired, the `handleComponentEvent` client-side controller action is called.

Event Handling Phases

Component event handlers are associated with the bubble phase by default. To add a handler for the capture phase instead, use the `phase` attribute.

```
<aura:handler name="sampleComponentEvent" event="ns:eventName"
    action="{!c.handleComponentEvent}" phase="capture" />
```

Get the Source of an Event

In the client-side controller action for an `<aura:handler>` tag, use `evt.getSource()` to find out which component fired the event, where `evt` is a reference to the event. To retrieve the source element, use `evt.getSource().getElement()`.

IN THIS SECTION:

Component Handling Its Own Event
A component can handle its own event by using the `<aura:handler>` tag in its markup.

Handle Component Event of Instantiated Component
A parent component can set a handler action when it instantiates a child component in its markup.

Handling Bubbled or Captured Component Events
Event propagation rules determine which components in the containment hierarchy can handle events by default in the bubble or capture phases. Learn about the rules and how to handle events in the bubble or capture phases.

Handling Component Events Dynamically
A component can have its handler bound dynamically via JavaScript. This is useful if a component is created in JavaScript on the client-side.

SEE ALSO:

Component Event Propagation
Handling Application Events

Component Handling Its Own Event

A component can handle its own event by using the `<aura:handler>` tag in its markup.

The `action` attribute of `<aura:handler>` sets the client-side controller action to handle the event. For example:

```
<aura:registerEvent name="sampleComponentEvent" type="c:compEvent"/>
<aura:handler name="sampleComponentEvent" event="c:compEvent"
    action="{!c.handleSampleEvent}"/>
```

Note: The `name` attributes in `<aura:registerEvent>` and `<aura:handler>` must match, since each event is defined by its name.

SEE ALSO:

Handle Component Event of Instantiated Component

Handle Component Event of Instantiated Component

A parent component can set a handler action when it instantiates a child component in its markup.

Let's a look at an example. `c:child` registers that it may fire a `sampleComponentEvent` event by using `<aura:registerEvent>` in its markup.

```
<!-- c:child -->
<aura:component>
    <aura:registerEvent name="sampleComponentEvent" type="c:compEvent"/>
</aura:component>
```

`c:parent` sets a handler for this event when it instantiates `c:child` in its markup.

```
<!-- parent.cmp -->
<aura:component>
    <c:child sampleComponentEvent="{!c.handleChildEvent}"/>
</aura:component>
```

Note how `c:parent` uses the following syntax to set a handler for the `sampleComponentEvent` event fired by `c:child`.

```
<c:child sampleComponentEvent="{!c.handleChildEvent}"/>
```

The syntax looks similar to how you set an attribute called `sampleComponentEvent`. However, in this case, `sampleComponentEvent` isn't an attribute. `sampleComponentEvent` matches the event name declared in `c:child`.

```
<aura:registerEvent name="sampleComponentEvent" type="c:compEvent"/>
```

The preceding syntax is a convenient shortcut for the normal way that a component declares a handler for an event. The parent component can only use this syntax to handle events from a direct descendent. If you want to be more explicit in `c:parent` that you're handling an event, or if the event might be fired by a component further down the component hierarchy, use an `<aura:handler>` tag instead of declaring the handler within the `<c:child>` tag.

```
<!-- parent.cmp -->
<aura:component>
    <aura:handler name="sampleComponentEvent" event="c:compEvent"
      action="{!c.handleSampleEvent}"/>
    <c:child />
</aura:component>
```

The two versions of `c:parent` markup behave the same. However, using `<aura:handler>` makes it more obvious that you're handling a `sampleComponentEvent` event.

SEE ALSO:

Component Handling Its Own Event

Handling Bubbled or Captured Component Events

Handling Bubbled or Captured Component Events

Event propagation rules determine which components in the containment hierarchy can handle events by default in the bubble or capture phases. Learn about the rules and how to handle events in the bubble or capture phases.

The framework supports *capture* and *bubble* phases for the propagation of component events. These phases are similar to DOM handling patterns and provide an opportunity for interested components to interact with an event and potentially control the behavior for subsequent handlers. The capture phase executes before the bubble phase.

Default Event Propagation Rules

By default, every parent in the containment hierarchy can't handle an event during the capture and bubble phases. Instead, the event propagates to every owner in the containment hierarchy.

A component's owner is the component that is responsible for its creation. For declaratively created components, the owner is the outermost component containing the markup that references the component firing the event. For programmatically created components, the owner component is the component that invoked `$A.createComponent` to create it.

The same rules apply for the capture phase, although the direction of event propagation (down) is the opposite of the bubble phase (up).

Confused? It makes more sense when you look at an example in the bubbling phase.

`c:owner` contains `c:container`, which in turn contains `c:eventSource`.

```
<!--c:owner-->
<aura:component>
    <c:container>
        <c:eventSource />
    </c:container>
</aura:component>
```

If `c:eventSource` fires an event, it can handle the event itself. The event then bubbles up the containment hierarchy.

`c:container` contains `c:eventSource` but it's not the owner because it's not the outermost component in the markup, so it can't handle the bubbled event.

`c:owner` is the owner because `c:container` is in its markup. `c:owner` can handle the event.

Propagation to All Container Components

The default behavior doesn't allow an event to be handled by every parent in the containment hierarchy. Some components contain other components but aren't the owner of those components. These components are known as container components. In the example, `c:container` is a container component because it's not the owner for `c:eventSource`. By default, `c:container` can't handle events fired by `c:eventSource`.

A container component has a facet attribute whose type is `Aura.Component[]`, such as the default `body` attribute. The container component includes those components in its definition using an expression, such as `{!v.body}`. The container component isn't the owner of the components rendered with that expression.

To allow a container component to handle the event, add `includeFacets="true"` to the `<aura:handler>` tag of the container component. For example, adding `includeFacets="true"` to the handler in the container component, `c:container`, enables it to handle the component event bubbled from `c:eventSource`.

```
<aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"
    includeFacets="true" />
```

Handle Bubbled Event

A component that fires a component event registers that it fires the event by using the `<aura:registerEvent>` tag.

```
<aura:component>
    <aura:registerEvent name="compEvent" type="c:compEvent" />
</aura:component>
```

A component handling the event in the bubble phase uses the `<aura:handler>` tag to assign a handling action in its client-side controller.

```
<aura:component>
    <aura:handler name="compEvent" event="c:compEvent" action="{!c.handleBubbling}"/>
</aura:component>
```

Note: The `name` attribute in `<aura:handler>` must match the `name` attribute in the `<aura:registerEvent>` tag in the component that fires the event.

Handle Captured Event

A component handling the event in the capture phase uses the `<aura:handler>` tag to assign a handling action in its client-side controller.

```
<aura:component>
    <aura:handler name="compEvent" event="c:compEvent" action="{!c.handleCapture}"
        phase="capture" />
</aura:component>
```

The default handling phase for component events is bubble if no `phase` attribute is set.

Stop Event Propagation

Use the `stopPropagation()` method in the `Event` object to stop the event propagating to other components.

Pausing Event Propagation for Asynchronous Code Execution

Use `event.pause()` to pause event handling and propagation until `event.resume()` is called. This flow-control mechanism is useful for any decision that depends on the response from the execution of asynchronous code. For example, you might make a decision about event propagation based on the response from an asynchronous call to native mobile code.

You can call `pause()` or `resume()` in the capture or bubble phases.

Event Bubbling Example

Let's look at an example so you can play around with it yourself.

```
<!--c:eventBubblingParent-->
<aura:component>
    <c:eventBubblingChild>
        <c:eventBubblingGrandchild />
    </c:eventBubblingChild>
</aura:component>
```

Note: This sample code uses the default `c` namespace. If your org has a namespace, use that namespace instead.

First, we define a simple component event.

```
<!--c:compEvent-->
<aura:event type="COMPONENT">
    <!--simple event with no attributes-->
</aura:event>
```

`c:eventBubblingEmitter` is the component that fires `c:compEvent`.

```
<!--c:eventBubblingEmitter-->
<aura:component>
    <aura:registerEvent name="bubblingEvent" type="c:compEvent" />
    <lightning:button onclick="{!c.fireEvent}" label="Start Bubbling"/>
</aura:component>
```

Here's the controller for `c:eventBubblingEmitter`. When you press the button, it fires the `bubblingEvent` event registered in the markup.

```
/*eventBubblingEmitterController.js*/
{
    fireEvent : function(cmp) {
        var cmpEvent = cmp.getEvent("bubblingEvent");
        cmpEvent.fire();
    }
}
```

`c:eventBubblingGrandchild` contains `c:eventBubblingEmitter` and uses `<aura:handler>` to assign a handler for the event.

```
<!--c:eventBubblingGrandchild-->
<aura:component>
    <aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"/>


    <div class="grandchild">
        <c:eventBubblingEmitter />
    </div>
</aura:component>
```

Here's the controller for `c:eventBubblingGrandchild`.

```
/*eventBubblingGrandchildController.js*/
{
    handleBubbling : function(component, event) {
        console.log("Grandchild handler for " + event.getName());
    }
}
```

The controller logs the event name when the handler is called.

Here's the markup for `c:eventBubblingChild`. We will pass `c:eventBubblingGrandchild` in as the body of `c:eventBubblingChild` when we create `c:eventBubblingParent` later in this example.

```
<!--c:eventBubblingChild-->
<aura:component>
    <aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"/>


    <div class="child">
        {!v.body}
    </div>
</aura:component>
```

Here's the controller for `c:eventBubblingChild`.

```
/*eventBubblingChildController.js*/
{
    handleBubbling : function(component, event) {
        console.log("Child handler for " + event.getName());
    }
}
```

`c:eventBubblingParent` contains `c:eventBubblingChild`, which in turn contains `c:eventBubblingGrandchild`.

```
<!--c:eventBubblingParent-->
<aura:component>
    <aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"/>


    <div class="parent">
        <c:eventBubblingChild>
            <c:eventBubblingGrandchild />
        </c:eventBubblingChild>
    </div>
</aura:component>
```

Here's the controller for `c:eventBubblingParent`.

```
/*eventBubblingParentController.js*/
{
    handleBubbling : function(component, event) {
        console.log("Parent handler for " + event.getName());
    }
}
```

Now, let's see what happens when you run the code.

1. In your browser, navigate to `c:eventBubblingParent`. Create a `.app` resource that contains `<c:eventBubblingParent />`.
2. Click the **Start Bubbling** button that is part of the markup in `c:eventBubblingEmitter`.

3. Note the output in your browser's console:

```
Grandchild handler for bubblingEvent
Parent handler for bubblingEvent
```

The `c:compEvent` event is bubbled to `c:eventBubblingGrandchild` and `c:eventBubblingParent` as they are owners in the containment hierarchy. The event is not handled by `c:eventBubblingChild` as `c:eventBubblingChild` is in the markup for `c:eventBubblingParent` but it's not an owner as it's not the outermost component in that markup.

Now, let's see how to stop event propagation. Edit the controller for `c:eventBubblingGrandchild` to stop propagation.

```
/*eventBubblingGrandchildController.js*/
{
    handleBubbling : function(component, event) {
        console.log("Grandchild handler for " + event.getName());
        event.stopPropagation();
    }
}
```

Now, navigate to `c:eventBubblingParent` and click the **Start Bubbling** button.

Note the output in your browser's console:

```
Grandchild handler for bubblingEvent
```

The event no longer bubbles up to the `c:eventBubblingParent` component.

SEE ALSO:

Component Event Propagation

Handle Component Event of Instantiated Component

Handling Component Events Dynamically

A component can have its handler bound dynamically via JavaScript. This is useful if a component is created in JavaScript on the client-side.

For more information, see Dynamically Adding Event Handlers To a Component on page 368.

Component Event Example

Here's a simple use case of using a component event to update an attribute in another component.

1. A user clicks a button in the notifier component, `ceNotifier.cmp`.
2. The client-side controller for `ceNotifier.cmp` sets a message in a component event and fires the event.
3. The handler component, `ceHandler.cmp`, contains the notifier component, and handles the fired event.
4. The client-side controller for `ceHandler.cmp` sets an attribute in `ceHandler.cmp` based on the data sent in the event.

Note: The event and components in this example use the default `c` namespace. If your org has a namespace, use that namespace instead.

Component Event

The `ceEvent.evt` component event has one attribute. We'll use this attribute to pass some data in the event when it's fired.

```
<!--c:ceEvent-->
<aura:event type="COMPONENT">
    <aura:attribute name="message" type="String"/>
</aura:event>
```

Notifier Component

The `c:ceNotifier` component uses `aura:registerEvent` to declare that it may fire the component event.

The button in the component contains an `onclick` browser event that is wired to the `fireComponentEvent` action in the client-side controller. The action is invoked when you click the button.

```
<!--c:ceNotifier-->
<aura:component>
    <aura:registerEvent name="cmpEvent" type="c:ceEvent"/>

    <h1>Simple Component Event Sample</h1>
    <p><lightning:button
        label="Click here to fire a component event"
        onclick="{!c.fireComponentEvent}" />
    </p>
</aura:component>
```

The client-side controller gets an instance of the event by calling `cmp.getEvent("cmpEvent")`, where `cmpEvent` matches the value of the name attribute in the `<aura:registerEvent>` tag in the component markup. The controller sets the `message` attribute of the event and fires the event.

```
/* ceNotifierController.js */
{
    fireComponentEvent : function(cmp, event) {
        // Get the component event by using the
        // name value from aura:registerEvent
        var cmpEvent = cmp.getEvent("cmpEvent");
        cmpEvent.setParams({
            "message" : "A component event fired me. " +
            "It all happened so fast. Now, I'm here!" });
        cmpEvent.fire();
    }
}
```

Handler Component

The `c:ceHandler` handler component contains the `c:ceNotifier` component. The `<aura:handler>` tag uses the same value of the `name` attribute, `cmpEvent`, from the `<aura:registerEvent>` tag in `c:ceNotifier`. This wires up `c:ceHandler` to handle the event bubbled up from `c:ceNotifier`.

When the event is fired, the `handleComponentEvent` action in the client-side controller of the handler component is invoked.

```
<!--c:ceHandler-->
<aura:component>
    <aura:attribute name="messageFromEvent" type="String"/>
    <aura:attribute name="numEvents" type="Integer" default="0"/>

    <!-- Note that name="cmpEvent" in aura:registerEvent
     in ceNotifier.cmp -->
    <aura:handler name="cmpEvent" event="c:ceEvent" action="{!c.handleComponentEvent}"/>

    <!-- handler contains the notifier component -->
    <c:ceNotifier />

    <p>{!v.messageFromEvent}</p>
    <p>Number of events: {!v.numEvents}</p>

</aura:component>
```

The controller retrieves the data sent in the event and uses it to update the `messageFromEvent` attribute in the handler component.

```
/* ceHandlerController.js */
{
    handleComponentEvent : function(cmp, event) {
        var message = event.getParam("message");

        // set the handler attributes based on event data
        cmp.set("v.messageFromEvent", message);
        var numEventsHandled = parseInt(cmp.get("v.numEvents")) + 1;
        cmp.set("v.numEvents", numEventsHandled);
    }
}
```

Put It All Together

Add the `c:ceHandler` component to a `c:ceHandlerApp` application. Navigate to the application and click the button to fire the component event.

`https://`*`MyDomainName`*`.lightning.force.com/c/ceHandlerApp.app`.

If you want to access data on the server, you could extend this example to call a server-side controller from the handler's client-side controller.

SEE ALSO:

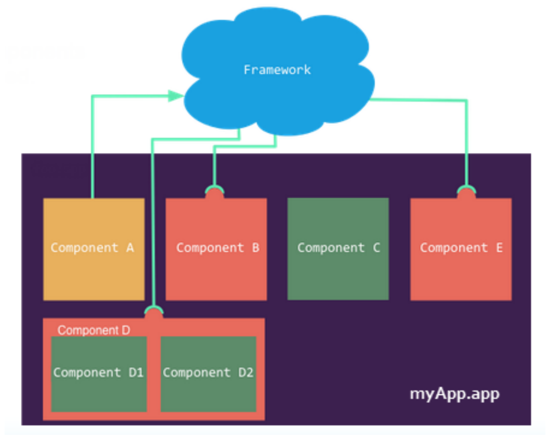
Component Events

Creating Server-Side Logic with Controllers

Application Event Example

Application Events

Application events follow a traditional publish-subscribe model. An application event is fired from an instance of a component. All components that provide a handler for the event are notified.



IN THIS SECTION:

SEE ALSO:

Component Events

Handling Events with Client-Side Controllers

Application Event Propagation

Advanced Events Example

Application Event Propagation

The framework supports *capture*, *bubble*, and *default* phases for the propagation of application events. The capture and bubble phases are similar to DOM handling patterns and provide an opportunity for interested components to interact with an event and potentially control the behavior for subsequent handlers. The default phase preserves the framework's original handling behavior.

A component can publish an application-level event. When the event is fired, any component or application that has subscribed to the event invokes its handler within a Lightning page. To communicate across the DOM within a Lightning page, or across multiple pages between Visualforce, Lightning pages, and Lightning web components (LWC), use Lightning Message Service on page 297 instead.

The component that fires an event is known as the source component. The framework allows you to handle the event in different phases. These phases give you flexibility for how to best process the event for your application.

The phases are:

Capture

The event is captured and trickles down from the application root to the source component. The event can be handled by a component in the containment hierarchy that receives the captured event.

Event handlers are invoked in order from the application root down to the source component that fired the event.

Any registered handler in this phase can stop the event from propagating, at which point no more handlers are called in this phase or the bubble phase. If a component stops the event propagation using `event.stopPropagation()`, the component becomes the root node used in the default phase.

Any registered handler in this phase can cancel the default behavior of the event by calling `event.preventDefault()`. This call prevents execution of any of the handlers in the default phase.

Bubble

The component that fired the event can handle it. The event then bubbles up from the source component to the application root. The event can be handled by a component in the containment hierarchy that receives the bubbled event.

Event handlers are invoked in order from the source component that fired the event up to the application root.

Any registered handler in this phase can stop the event from propagating, at which point no more handlers will be called in this phase. If a component stops the event propagation using `event.stopPropagation()`, the component becomes the root node used in the default phase.

Any registered handler in this phase can cancel the default behavior of the event by calling `event.preventDefault()`. This call prevents execution of any of the handlers in the default phase.

Default

Event handlers are invoked in a non-deterministic order from the root node through its subtree. The default phase doesn't have the same propagation rules related to component hierarchy as the capture and bubble phases. The default phase can be useful for handling application events that affect components in different sub-trees of your app.

If the event's propagation wasn't stopped in a previous phase, the root node defaults to the application root. If the event's propagation was stopped in a previous phase, the root node is set to the component whose handler invoked `event.stopPropagation()`.

Here is the sequence of application event propagation.

1. **Event fired**—An application event is fired. The component that fires the event is known as the source component.
2. **Capture phase**—The framework executes the capture phase from the application root to the source component until all components are traversed. Any handling event can stop propagation by calling `stopPropagation()` on the event.
3. **Bubble phase**—The framework executes the bubble phase from the source component to the application root until all components are traversed or `stopPropagation()` is called.
4. **Default phase**—The framework executes the default phase from the root node unless `preventDefault()` was called in the capture or bubble phases. If the event's propagation wasn't stopped in a previous phase, the root node defaults to the application root. If the event's propagation was stopped in a previous phase, the root node is set to the component whose handler invoked `event.stopPropagation()`.

Create Custom Application Events

Create a custom application event using the `<aura:event>` tag in a `.evt` resource. Events can contain attributes that can be set before the event is fired and read when the event is handled.

Use `type="APPLICATION"` in the `<aura:event>` tag for an application event. For example, this `c:appEvent` application event has one attribute with a name of `message`.

```
<!--c:appEvent-->
<aura:event type="APPLICATION">
    <!-- Add aura:attribute tags to define event shape.
         One sample attribute here. -->
    <aura:attribute name="message" type="String"/>
</aura:event>
```

The component that fires an event can set the event's data. To set the attribute values, call `event.setParam()` or `event.setParams()`. A parameter name set in the event must match the `name` attribute of an `<aura:attribute>` in the event. For example, if you fire `c:appEvent`, you could use:

```
event.setParam("message", "event message here");
```

The component that handles an event can retrieve the event data. To retrieve the attribute in this event, call `event.getParam("message")` in the handler's client-side controller.

SEE ALSO:

Application Event Example

Fire Application Events

Application events follow a traditional publish-subscribe model. An application event is fired from an instance of a component. All components that provide a handler for the event are notified.

Register an Event

A component registers that it may fire an application event by using `<aura:registerEvent>` in its markup. The `name` attribute is required but not used for application events. The `name` attribute is only relevant for component events. This example uses `name="appEvent"` but the value isn't used anywhere.

```
<aura:registerEvent name="appEvent" type="c:appEvent"/>
```

Fire an Event

Use `$A.get("e.myNamespace:myAppEvent")` in JavaScript to get an instance of the `myAppEvent` event in the `myNamespace` namespace.

Note: The syntax to get an instance of an application event is different than the syntax to get a component event, which is `cmp.getEvent("`***`evtName`***`")`.

Use `fire()` to fire the event.

```
var appEvent = $A.get("e.c:appEvent");
// Optional: set some data for the event (also known as event shape)
// A parameter's name must match the name attribute
// of one of the event's <aura:attribute> tags
//appEvent.setParams({ "myParam" : myValue });
appEvent.fire();
```

Events Fired on App Rendering

Some events are automatically fired when an app is rendering. For more information, see Events Fired During the Rendering Lifecycle on page 292.

SEE ALSO:

Fire Component Events

Handling Application Events

Use `<aura:handler>` in the markup of the handler component.

For example:

```
<aura:handler event="c:appEvent" action="{!c.handleApplicationEvent}"/>
```

The `event` attribute specifies the event being handled. The format is ***`namespace:eventName`***.

The `action` attribute of `<aura:handler>` sets the client-side controller action to handle the event.

Note: The handler for an application event won't work if you set the `name` attribute in `<aura:handler>`. Use the `name` attribute only when you're handling component events.

In this example, when the event is fired, the `handleApplicationEvent` client-side controller action is called.

Event Handling Phases

The framework allows you to handle the event in different phases. These phases give you flexibility for how to best process the event for your application.

Application event handlers are associated with the default phase. To add a handler for the capture or bubble phases instead, use the `phase` attribute.

Get the Source of an Event

In the client-side controller action for an `<aura:handler>` tag, use `evt.getSource()` to find out which component fired the event, where `evt` is a reference to the event. To retrieve the source element, use `evt.getSource().getElement()`.

IN THIS SECTION:

Handling Bubbled or Captured Application Events

Event propagation rules determine which components in the containment hierarchy can handle events by default in the bubble or capture phases. Learn about the rules and how to handle events in the bubble or capture phases.

SEE ALSO:

Handling Component Events

Handling Bubbled or Captured Application Events

Event propagation rules determine which components in the containment hierarchy can handle events by default in the bubble or capture phases. Learn about the rules and how to handle events in the bubble or capture phases.

The framework supports *capture*, *bubble*, and *default* phases for the propagation of application events. The capture and bubble phases are similar to DOM handling patterns and provide an opportunity for interested components to interact with an event and potentially control the behavior for subsequent handlers. The default phase preserves the framework's original handling behavior.

Default Event Propagation Rules

By default, every parent in the containment hierarchy can't handle an event during the capture and bubble phases. Instead, the event propagates to every owner in the containment hierarchy.

A component's owner is the component that is responsible for its creation. For declaratively created components, the owner is the outermost component containing the markup that references the component firing the event. For programmatically created components, the owner component is the component that invoked `$A.createComponent` to create it.

The same rules apply for the capture phase, although the direction of event propagation (down) is the opposite of the bubble phase (up).

Confused? It makes more sense when you look at an example in the bubbling phase.

`c:owner` contains `c:container`, which in turn contains `c:eventSource`.

```
<!--c:owner-->
<aura:component>
    <c:container>
        <c:eventSource />
    </c:container>
</aura:component>
```

If `c:eventSource` fires an event, it can handle the event itself. The event then bubbles up the containment hierarchy.

`c:container` contains `c:eventSource` but it's not the owner because it's not the outermost component in the markup, so it can't handle the bubbled event.

`c:owner` is the owner because `c:container` is in its markup. `c:owner` can handle the event.

Propagation to All Container Components

The default behavior doesn't allow an event to be handled by every parent in the containment hierarchy. Some components contain other components but aren't the owner of those components. These components are known as container components. In the example, `c:container` is a container component because it's not the owner for `c:eventSource`. By default, `c:container` can't handle events fired by `c:eventSource`.

A container component has a facet attribute whose type is `Aura.Component[]`, such as the default `body` attribute. The container component includes those components in its definition using an expression, such as `{!v.body}`. The container component isn't the owner of the components rendered with that expression.

To allow a container component to handle the event, add `includeFacets="true"` to the `<aura:handler>` tag of the container component. For example, adding `includeFacets="true"` to the handler in the container component, `c:container`, enables it to handle the component event bubbled from `c:eventSource`.

```
<aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"
    includeFacets="true" />
```

Handle Bubbled Event

To add a handler for the bubble phase, set `phase="bubble"`.

```
<aura:handler event="c:appEvent" action="{!c.handleBubbledEvent}"
    phase="bubble" />
```

The `event` attribute specifies the event being handled. The format is ***`namespace:eventName`***.

The `action` attribute of `<aura:handler>` sets the client-side controller action to handle the event.

Handle Captured Event

To add a handler for the capture phase, set `phase="capture"`.

```
<aura:handler event="c:appEvent" action="{!c.handleCapturedEvent}"
    phase="capture" />
```

Stop Event Propagation

Use the `stopPropagation()` method in the `Event` object to stop the event propagating to other components.

Pausing Event Propagation for Asynchronous Code Execution

Use `event.pause()` to pause event handling and propagation until `event.resume()` is called. This flow-control mechanism is useful for any decision that depends on the response from the execution of asynchronous code. For example, you might make a decision about event propagation based on the response from an asynchronous call to native mobile code.

You can call `pause()` or `resume()` in the capture or bubble phases.

Application Event Example

Here's a simple use case of using an application event to update an attribute in another component.

1. A user clicks a button in the notifier component, `aeNotifier.cmp`.
2. The client-side controller for `aeNotifier.cmp` sets a message in a component event and fires the event.
3. The handler component, `aeHandler.cmp`, handles the fired event.
4. The client-side controller for `aeHandler.cmp` sets an attribute in `aeHandler.cmp` based on the data sent in the event.

Note: The event and components in this example use the default `c` namespace. If your org has a namespace, use that namespace instead.

Application Event

The `aeEvent.evt` application event has one attribute. We'll use this attribute to pass some data in the event when it's fired.

```
<!--c:aeEvent-->
<aura:event type="APPLICATION">
    <aura:attribute name="message" type="String"/>
</aura:event>
```

Notifier Component

The `aeNotifier.cmp` notifier component uses `aura:registerEvent` to declare that it may fire the application event. The `name` attribute is required but not used for application events. The `name` attribute is only relevant for component events.

The button in the component contains a `onclick` browser event that is wired to the `fireApplicationEvent` action in the client-side controller. Clicking this button invokes the action.

```
<!--c:aeNotifier-->
<aura:component>
    <aura:registerEvent name="appEvent" type="c:aeEvent"/>

    <h1>Simple Application Event Sample</h1>
    <p><lightning:button
        label="Click here to fire an application event"
        onclick="{!c.fireApplicationEvent}" />
    </p>
</aura:component>
```

The client-side controller gets an instance of the event by calling `$A.get("e.c:aeEvent")`. The controller sets the `message` attribute of the event and fires the event.

```
/* aeNotifierController.js */
{
    fireApplicationEvent : function(cmp, event) {
        // Get the application event by using the
        // e.<namespace>.<event> syntax
        var appEvent = $A.get("e.c:aeEvent");
        appEvent.setParams({
            "message" : "An application event fired me. " +
            "It all happened so fast. Now, I'm everywhere!" });
        appEvent.fire();
    }
}
```

Handler Component

The `aeHandler.cmp` handler component uses the `<aura:handler>` tag to register that it handles the application event.

Note: The handler for an application event won't work if you set the `name` attribute in `<aura:handler>`. Use the `name` attribute only when you're handling component events.

When the event is fired, the `handleApplicationEvent` action in the client-side controller of the handler component is invoked.

```
<!--c:aeHandler-->
<aura:component>
    <aura:attribute name="messageFromEvent" type="String"/>
    <aura:attribute name="numEvents" type="Integer" default="0"/>

    <aura:handler event="c:aeEvent" action="{!c.handleApplicationEvent}"/>

    <p>{!v.messageFromEvent}</p>
    <p>Number of events: {!v.numEvents}</p>
</aura:component>
```

The controller retrieves the data sent in the event and uses it to update the `messageFromEvent` attribute in the handler component.

```
/* aeHandlerController.js */
{
    handleApplicationEvent : function(cmp, event) {
        var message = event.getParam("message");

        // set the handler attributes based on event data
        cmp.set("v.messageFromEvent", message);
        var numEventsHandled = parseInt(cmp.get("v.numEvents")) + 1;
        cmp.set("v.numEvents", numEventsHandled);
    }
}
```

Container Component

The `aeContainer.cmp` container component contains the notifier and handler components. This is different from the component event example where the handler contains the notifier component.

```
<!--c:aeContainer-->
<aura:component>
    <c:aeNotifier/>
    <c:aeHandler/>
</aura:component>
```

Put It All Together

You can test this code by adding `<c:aeContainer>` to a sample `aeWrapper.app` application and navigating to the application.

`https://`*`MyDomainName`*`.lightning.force.com/c/aeWrapper.app`.

If you want to access data on the server, you could extend this example to call a server-side controller from the handler's client-side controller.

SEE ALSO:

- Application Events
- Creating Server-Side Logic with Controllers
- Component Event Example

Event Handler Behavior for Active Components

To prevent active event handlers on cached pages from causing problems, add a workaround to your code to check if the component is still visible. To avoid this scenario and the workaround, use Lightning message service instead to communicate across the DOM within a Lightning page. The default scope used by Lightning message service channels publishes only to active components.

When navigating away from a page in Lightning Experience, the framework caches the components in the page so that they remain active, along with their event handlers. This caching speeds up navigation, but it can cause the cached component to respond to events that are not intended for it, such as `force:refreshView` or `force:recordSaveSuccess`.

This workaround uses the `offsetParent` property for the component to get its handlers while they're visible. The workaround is good only if the component definition has an HTML element in it.

This component includes an event handler and some HTML.

```
<!--myComponent.cmp-->
<aura:component>
  <aura:handler event="c:appEvent" action="{!c.onEvent}>
  <h1>This component has a handler</h1>
</aura:component>
```

Here's the client-side controller that uses the `offsetParent` property to get the component's handlers while they're still visible.

```
/* myComponentController.js */
({
  onEvent: function(component, event, helper) {
    var elem = component.getElement();
    if (elem && elem.offsetParent !== null) {
      // event handling logic here
    }
  }
})
```

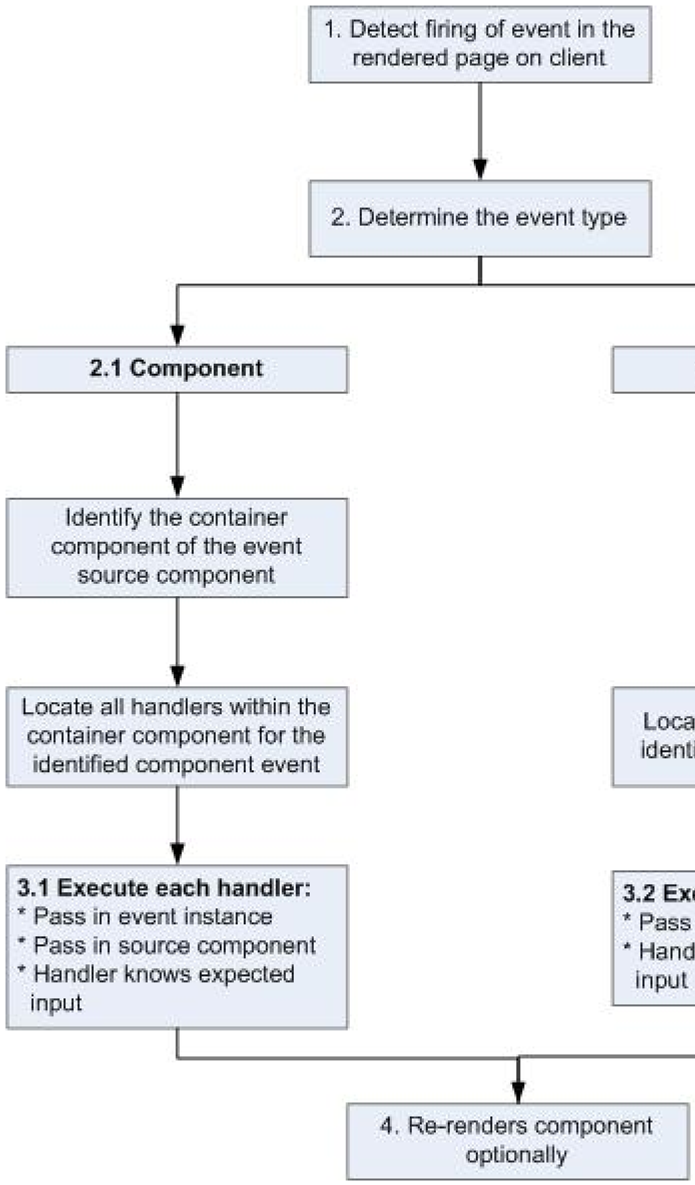
SEE ALSO:

Communicating Across the DOM with Lightning Message Service

Component Library: Message Service

Event Handling Lifecycle

The following chart summarizes how the framework handles events.


1. Detect firing of event in the rendered page on client
2. Determine the event type
2.1 Component
2.2 Application
Identify the container component of the event source component
Locate all handlers within the container component for the identified component event
Locate all handlers for the identified application event
3.1 Execute each handler:
* Pass in event instance
* Pass in source component
* Handler knows expected input
3.2 Execute each handler:
* Pass in event instance
* Handler knows expected input
4. Re-renders component optionally

1 Detect Firing of Event

The framework detects the firing of an event. For example, the event could be triggered by a button click in a notifier component.

2 Determine the Event Type

2.1 Component Event

The parent or container component instance that fired the event is identified. This container component locates all relevant event handlers for further processing.

2.2 Application Event

Any component can have an event handler for this event. All relevant event handlers are located.

3 Execute each Handler

3.1 Executing a Component Event Handler

Each of the event handlers defined in the container component for the event are executed by the handler controller, which can also:

- Set attributes or modify data on the component (causing a re-rendering of the component).
- Fire another event or invoke a client-side or server-side action.

3.2 Executing an Application Event Handler

All event handlers are executed. When the event handler is executed, the event instance is passed into the event handler.

4 Re-render Component (optional)

After the event handlers and any callback actions are executed, a component might be automatically re-rendered if it was modified during the event handling process.

SEE ALSO:

Create a Custom Renderer

Advanced Events Example

This example builds on the simpler component and application event examples. It uses one notifier component and one handler component that work with both component and application events. Before we see a component wired up to events, let's look at the individual resources involved.

This table summarizes the roles of the various resources used in the example. The source code for these resources is included after the table.

Resource	Resource Name	Usage
Event files	Component event (`compEvent.evt`) and application event (`appEvent.evt`)	Defines the component and application events in separate resources. `eventsContainer.cmp` shows how to use both component and application events.
Notifier	Component (`eventsNotifier.cmp`) and its controller (`eventsNotifierController.js`)	The notifier contains an `onclick` browser event to initiate the event. The controller fires the event.
Handler	Component (`eventsHandler.cmp`) and its controller (`eventsHandlerController.js`)	The handler component contains the notifier component (or a `<aura:handler>` tag for application events), and calls the controller action that is executed after the event is fired.
Container Component	`eventsContainer.cmp`	Displays the event handlers on the UI for the complete demo.

The definitions of component and application events are stored in separate `.evt` resources, but individual notifier and handler component bundles can contain code to work with both types of events.

The component and application events both contain a `context` attribute that defines the shape of the event. This is the data that is passed to handlers of the event.

Component Event

Here is the markup for `compEvent.evt`.

```
<!--c:compEvent-->
<aura:event type="COMPONENT">
    <!-- pass context of where the event was fired to the handler. -->
    <aura:attribute name="context" type="String"/>
</aura:event>
```

Application Event

Here is the markup for `appEvent.evt`.

```
<!--c:appEvent-->
<aura:event type="APPLICATION">
    <!-- pass context of where the event was fired to the handler. -->
    <aura:attribute name="context" type="String"/>
</aura:event>
```

Notifier Component

The `eventsNotifier.cmp` notifier component contains buttons to initiate a component or application event.

The notifier uses `aura:registerEvent` tags to declare that it may fire the component and application events. Note that the `name` attribute is required but the value is only relevant for the component event; the value is not used anywhere else for the application event.

The `parentName` attribute is not set yet. We will see how this attribute is set and surfaced in `eventsContainer.cmp`.

```
<!--c:eventsNotifier-->
<aura:component>
  <aura:attribute name="parentName" type="String"/>
  <aura:registerEvent name="componentEventFired" type="c:compEvent"/>
  <aura:registerEvent name="appEvent" type="c:appEvent"/>

  <div>
    <h3>This is {!v.parentName}'s eventsNotifier.cmp instance</h3>
    <p><lightning:button
        label="Click here to fire a component event"
        onclick="{!c.fireComponentEvent}" />
    </p>
    <p><lightning:button
        label="Click here to fire an application event"
        onclick="{!c.fireApplicationEvent}" />
    </p>
  </div>
</aura:component>
```

CSS source

The CSS is in `eventsNotifier.css`.

```
/* eventsNotifier.css */
.cEventsNotifier {
    display: block;
    margin: 10px;
    padding: 10px;
    border: 1px solid black;
}
```

Client-side controller source

The `eventsNotifierController.js` controller fires the event.

```
/* eventsNotifierController.js */
{
    fireComponentEvent : function(cmp, event) {
        var parentName = cmp.get("v.parentName");

        // Look up event by name, not by type
        var compEvents = cmp.getEvent("componentEventFired");

        compEvents.setParams({ "context" : parentName });
        compEvents.fire();
    },

    fireApplicationEvent : function(cmp, event) {
        var parentName = cmp.get("v.parentName");

        // note different syntax for getting application event
        var appEvent = $A.get("e.c:appEvent");

        appEvent.setParams({ "context" : parentName });
        appEvent.fire();
    }
}
```

You can click the buttons to fire component and application events but there is no change to the output because we haven't wired up the handler component to react to the events yet.

The controller sets the `context` attribute of the component or application event to the `parentName` of the notifier component before firing the event. We will see how this affects the output when we look at the handler component.

Handler Component

The `eventsHandler.cmp` handler component contains the `c:eventsNotifier` notifier component and `<aura:handler>` tags for the application and component events.

```
<!--c:eventsHandler-->
<aura:component>
  <aura:attribute name="name" type="String"/>
  <aura:attribute name="mostRecentEvent" type="String" default="Most recent event handled:"/>

  <aura:attribute name="numComponentEventsHandled" type="Integer" default="0"/>
  <aura:attribute name="numApplicationEventsHandled" type="Integer" default="0"/>

  <aura:handler event="c:appEvent" action="{!c.handleApplicationEventFired}"/>
  <aura:handler name="componentEventFired" event="c:compEvent"
action="{!c.handleComponentEventFired}"/>
```

```
  <div>
    <h3>This is {!v.name}</h3>
    <p>{!v.mostRecentEvent}</p>
    <p># component events handled: {!v.numComponentEventsHandled}</p>
    <p># application events handled: {!v.numApplicationEventsHandled}</p>
    <c:eventsNotifier parentName="{#v.name}" />
  </div>
</aura:component>
```

Note: `{#v.name}` is an unbound expression. This means that any change to the value of the `parentName` attribute in `c:eventsNotifier` doesn't propagate back to affect the value of the `name` attribute in `c:eventsHandler`. For more information, see Data Binding Between Components on page 48.

CSS source

The CSS is in `eventsHandler.css`.

```
/* eventsHandler.css */
.cEventsHandler {
  display: block;
  margin: 10px;
  padding: 10px;
  border: 1px solid black;
}
```

Client-side controller source

The client-side controller is in `eventsHandlerController.js`.

```
/* eventsHandlerController.js */
{
    handleComponentEventFired : function(cmp, event) {
        var context = event.getParam("context");
        cmp.set("v.mostRecentEvent",
            "Most recent event handled: COMPONENT event, from " + context);

        var numComponentEventsHandled =
            parseInt(cmp.get("v.numComponentEventsHandled")) + 1;
        cmp.set("v.numComponentEventsHandled", numComponentEventsHandled);
    },

    handleApplicationEventFired : function(cmp, event) {
        var context = event.getParam("context");
        cmp.set("v.mostRecentEvent",
            "Most recent event handled: APPLICATION event, from " + context);

        var numApplicationEventsHandled =
            parseInt(cmp.get("v.numApplicationEventsHandled")) + 1;
        cmp.set("v.numApplicationEventsHandled", numApplicationEventsHandled);
    }
}
```

The `name` attribute is not set yet. We will see how this attribute is set and surfaced in `eventsContainer.cmp`.

You can click buttons and the UI now changes to indicate the type of event. The click count increments to indicate whether it's a component or application event. We aren't finished yet though. Notice that the source of the event is undefined as the event `context` attribute hasn't been set .

Container Component

Here is the markup for `eventsContainer.cmp`.

```
<!--c:eventsContainer-->
<aura:component>
    <c:eventsHandler name="eventsHandler1"/>
    <c:eventsHandler name="eventsHandler2"/>
</aura:component>
```

The container component contains two handler components. It sets the `name` attribute of both handler components, which is passed through to set the `parentName` attribute of the notifier components. This fills in the gaps in the UI text that we saw when we looked at the notifier or handler components directly.

Add the `c:eventsContainer` component to a `c:eventsContainerApp` application. Navigate to the application.

`https://`*`MyDomainName`*`.lightning.force.com/c/eventsContainerApp.app`.

Click the **Click here to fire a component event** button for either of the event handlers. Notice that the **# component events handled** counter only increments for that component because only the firing component's handler is notified.

Click the **Click here to fire an application event** button for either of the event handlers. Notice that the **# application events handled** counter increments for both the components this time because all the handling components are notified.

SEE ALSO:

- Component Event Example
- Application Event Example
- Event Handling Lifecycle

Firing Events from Non-Aura Code

You can fire Aura events from JavaScript code outside an Aura app. For example, your Aura app might need to call out to some non-Aura code, and then have that code communicate back to your Aura app once it's done.

For example, you could call external code that needs to log into another system and return some data to your Aura app by firing an Aura event. Let's call this event `mynamespace:externalEvent`. The external code fires this event when it’s ready to communicate with an Aura app.

```
var myExternalEvent;
if(window.$A &&
   (myExternalEvent = window.$A.get("e.mynamespace:externalEvent"))) {
     myExternalEvent.setParams({isOauthed:true});
```

```
    myExternalEvent.fire();
}
```

SEE ALSO:

Application Events

Modifying Components Outside the Framework Lifecycle

Events Best Practices

Here are some best practices for working with events.

Use Component Events Whenever Possible

Always try to use a component event instead of an application event, if possible. Component events can only be handled by components above them in the containment hierarchy so their usage is more localized to the components that need to know about them. Application events are best used for something that should be handled at the application level, such as navigating to a specific record. Application events allow communication between components that are in separate parts of the application and have no direct containment relationship.

Separate Low-Level Events from Business Logic Events

Handle low-level events, such as a click, in your event handler and refire them as higher-level events, such as an `approvalChange` event or whatever is appropriate for your business logic.

Dynamic Actions Based on Component State

To invoke a different action on a click event depending on the state of the component, try this approach:

1. Store the component state as a discrete value, such as New or Pending, in a component attribute.
2. Put logic in your client-side controller that determines the next action to take.
3. Put logic in the helper if you want to reuse it in the component bundle.

For example:

1. Your component markup contains `<lightning:button label="do something" onclick="{!c.handleClick}" />`.
2. In your controller, define the `handleClick` function, which delegates to the appropriate helper function or potentially fires the correct event.

Using a Dispatcher Component to Listen and Relay Events

If you have a large number of handler component instances listening for an event, identify a dispatcher component to listen for the event. The dispatcher component can perform some logic to decide which component instances receive further information, and fire another component or application event targeted at those component instances.

SEE ALSO:

Handling Events with Client-Side Controllers

Events Anti-Patterns

Events Anti-Patterns

These are some anti-patterns that you should avoid when using events.

Don't Fire an Event in a Renderer

Firing an event in a renderer can cause an infinite rendering loop.

Don't do this!

```
afterRender: function(cmp, helper) {
    this.superAfterRender();
    $A.get("e.myns:mycmp").fire();
}
```

Instead, use the `init` hook to run a controller action after component construction but before rendering. Add this code to your component:

```
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
```

For more details, see .Invoking Actions on Component Initialization on page 337.

Don't Use `onclick` and `ontouchend` Events

You can't use different actions for `onclick` and `ontouchend` events in a component. The framework translates touch-tap events into clicks and activates any `onclick` handlers that are present.

SEE ALSO:

Create a Custom Renderer

Events Best Practices

Events Fired During the Rendering Lifecycle

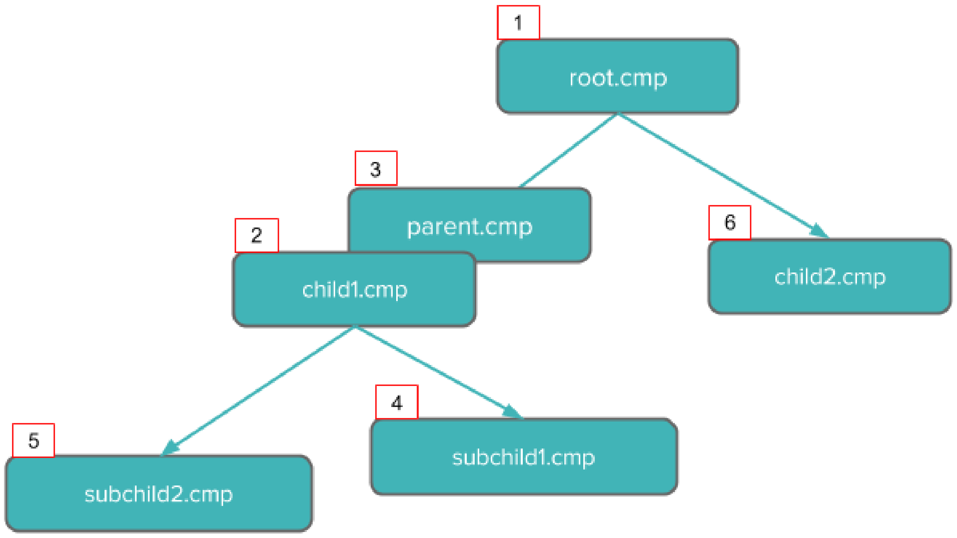
A component is instantiated, rendered, and rerendered during its lifecycle. A component rerenders only when there's a programmatic or value change that requires a rerender. For example, if a browser event triggers an action that updates the component's data, the component rerenders.

Component Creation

The component lifecycle starts when the client sends an HTTP request to the server and the component configuration data is returned to the client. No server trip is made if the component definition is already on the client from a previous request and the component has no server dependencies.

Let's look at an app with several nested components. The framework instantiates the app and goes through the children of the `v.body` facet to create each component. First, it creates the component definition, its entire parent hierarchy, and then creates the facets within those components. The framework also creates any component dependencies on the server, including definitions for attributes, interfaces, controllers, and actions.

The following image lists the order of component creation.



After creating a component instance, the framework sends the serialized component definitions and instances down to the client. Definitions are cached but not the instance data. The client deserializes the response to create the JavaScript objects or maps, resulting in an instance tree that's used to render the component instance. When the component tree is ready, the `init` event is fired for all the components, starting from the child components and finishing in the parent component.

Component Rendering

The rendering lifecycle happens once in the lifetime of a component unless the component gets explicitly unrendered. When you create a component:

1. The component service that constructs the components fires the `init` event to signal that initialization has completed.

   ```
   <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
   ```

   You can customize the `init` handler and add your own controller logic before the component starts rendering. For more information, see Invoking Actions on Component Initialization on page 337.

2. For each component in the tree, the base implementation of `render()` or your custom renderer is called to start component rendering. For more information, see Create a Custom Renderer on page 355. Similar to the component creation process, rendering starts at the root component, its child components and their super components, if any, and finally the subchild components.

3. After your components are rendered to the DOM, `afterRender()` is called to signal that rendering is completed for each of these component definitions. It enables you to interact with the DOM tree after the framework rendering service has created the DOM elements.
4. To indicate that the client is done waiting for a response to the server request XHR, the `aura:doneWaiting` event is fired. You can handle this event by adding a handler wired to a client-side controller action.

   Note: The `aura:doneWaiting` event is deprecated. The `aura:doneWaiting` application event is fired for every server response, even for responses from other components in your app. Unless your component is running in complete isolation in a standalone app and not included in Lightning Experience or the Salesforce mobile app, the container app may trigger your event handler multiple times. This behavior makes it difficult to handle each event appropriately.

5. The framework fires a `render` event, enabling you to interact with the DOM tree after the framework's rendering service has inserted DOM elements. Handling the `render` event is preferred to creating a custom renderer and overriding `afterRender()`. For more information, see Handle the render Event.
6. Finally, the `aura:doneRendering` event is fired at the end of the rendering lifecycle.

   Note: The `aura:doneRendering` event is deprecated. Unless your component is running in complete isolation in a standalone app and not included in complex apps, such as Lightning Experience or the Salesforce mobile app, the container app may trigger your event handler multiple times. This behavior makes it difficult to handle each event appropriately.

Rendering Nested Components

Let's say that you have an app `myApp.app` that contains a component `myCmp.cmp` with a nested component.

During initialization, the `init()` event is fired in this order: the nested component, `myCmp.cmp`, and `myApp.app`.

SEE ALSO:

Create a Custom Renderer

Events Handled in the Salesforce Mobile App and Lightning Experience

The Salesforce mobile app and Lightning Experience handle some events, which you can fire in your Aura component.

If you fire one of these `force` or `lightning` events in your Lightning apps or components outside of the Salesforce mobile app or Lightning Experience:

- You must handle the event by using the `<aura:handler>` tag in the handling component.
- Use the `<aura:registerEvent>` or `<aura:dependency>` tags to ensure that the event is sent to the client, when needed.

Event Name	Description
`force:closeQuickAction`	Closes a quick action panel. Only one quick action panel can be open in the app at a time.
`force:createRecord`	Opens a page to create a record for the specified `entityApiName`, for example, "Account" or "myNamespace__MyObject__c".
`force:editRecord`	Opens the page to edit the record specified by `recordId`.
`force:navigateToComponent` (Beta)	Navigates from one Aura component to another.

Event Name	Description
`force:navigateToList`	Navigates to the list view specified by `listViewId`.
`force:navigateToObjectHome`	Navigates to the object home specified by the `scope` attribute.
`force:navigateToRelatedList`	Navigates to the related list specified by `parentRecordId`.
`force:navigateToSObject`	Navigates to an sObject record specified by `recordId`.
`force:navigateToURL`	Navigates to the specified URL.
`force:recordSave`	Saves a record.
`force:recordSaveSuccess`	Indicates that the record has been successfully saved.
`force:refreshView`	Reloads the view.
`force:showToast`	Displays a toast notification with a message. (Not available on login pages.)
`lightning:openFiles`	Opens one or more file records from the ContentDocument and ContentHubItem objects.

Customizing Client-Side Logic for the Salesforce Mobile App, Lightning Experience, and Standalone Apps

Since the Salesforce mobile app and Lightning Experience automatically handle many events, you have to do extra work if your component runs in a standalone app. Instantiating the event using `$A.get()` can help you determine if your component is running within the Salesforce mobile app and Lightning Experience or a standalone app. For example, you want to display a toast when a component loads in the Salesforce mobile app and Lightning Experience. You can fire the `force:showToast` event and set its parameters for the Salesforce mobile app and Lightning Experience, but you have to create your own implementation for a standalone app.

```
displayToast : function (component, event, helper) {
    var toast = $A.get("e.force:showToast");
    if (toast){
        //fire the toast event in Salesforce app and Lightning Experience
        toast.setParams({
            "title": "Success!",
            "message": "The component loaded successfully."
        });
        toast.fire();
    } else {
        //your toast implementation for a standalone app here
    }
}
```

SEE ALSO:

- aura:dependency
- Fire Component Events
- Fire Application Events

System Events

The framework fires several system events during its lifecycle.

You can handle these events in your Lightning apps or components, and within the Salesforce mobile app.

For examples, see the Component Library.

Event Name	Description
`aura:doneRendering` (deprecated)	Indicates that the initial rendering of the root application has completed. Note: The `aura:doneRendering` event is deprecated. Unless your component is running in complete isolation in a standalone app and not included in complex apps, such as Lightning Experience or the Salesforce mobile app, the container app may trigger your event handler multiple times. This behavior makes it difficult to handle each event appropriately.
`aura:doneWaiting` (deprecated)	Indicates that the app is done waiting for a response to a server request. This event is preceded by an `aura:waiting` event. Note: The `aura:doneWaiting` event is deprecated. The `aura:doneWaiting` application event is fired for every server response, even for responses from other components in your app. Unless your component is running in complete isolation in a standalone app and not included in Lightning Experience or the Salesforce mobile app, the container app may trigger your event handler multiple times. This behavior makes it difficult to handle each event appropriately.
`aura:locationChange`	Indicates that the hash part of the URL has changed.
`aura:noAccess`	Indicates that a requested resource is not accessible due to security constraints on that resource.
`aura:systemError`	Indicates that an error has occurred.
`aura:valueChange`	Indicates that an attribute value has changed.
`aura:valueDestroy`	Indicates that a component has been destroyed.
`aura:valueInit`	Indicates that an app or component has been initialized.
`aura:valueRender`	Indicates that an app or component has been rendered or rerendered.
`aura:waiting` (deprecated)	Indicates that the app is waiting for a response to a server request. Note: The `aura:waiting` event is deprecated. The `aura:waiting` application event is fired for every server request, even for requests from other components in your app. Unless your component is running in complete isolation in a standalone app and not included in Lightning Experience or the Salesforce mobile app, the container app may trigger your event handler multiple times. This behavior makes it difficult to handle each event appropriately.

CHAPTER 6 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 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 Aura, use the `lightning:messageChannel` component. A message is a serializable JSON object. Examples of data that you can pass in a message include strings, numbers, booleans, and objects. A message can't contain functions and symbols. The `lightning:messageChannel` component is only available in Lightning Experience.

SEE ALSO:

Blog: Lightning Message Service

Lightning Web Components Developer Guide: Communicating Across the DOM with Lightning Message Service

Visualforce 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:messageChannel` component in your org, use the LightningMessageChannel metadata type and append it with `__c`. The message channel isn't a custom object, it just uses the same suffix.

Note: See LightningMessageChannel in the Metadata API Developer Guide.

To deploy a LightningMessageChannel into your org, create a 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 deploy 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 a message on a message channel, include a `lightning:messageChannel` component in your Aura component and use the `publish()` method in your Aura component's controller file.

Example: The `lmsPublisherAuraComponent` 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.

To reference a message channel, add the `lightning:messageChannel` component to your Aura component. The component has a required `type` attribute, which is the name of the message channel.

```
<!-- myComponent.cmp -->
<aura:component>
    <lightning:messageChannel type="SampleMessageChannel__c"/>
</aura:component>
```

To reference a message channel from an org that has a namespace, prefix the message channel name with the namespace: `<lightning:messageChannel type="`***`Namespace__MessageChannelName__c`***`"/>`.

This example shows how to publish a message on the `SampleMessageChannel__c` channel when a button is clicked.

In `myComponent.cmp`, we create two components, `lightning:button` and `lightning:messageChannel`. On `lightning:button`, the `onclick` handler calls the `handleClick()` JavaScript function in the controller. We assign the `aura:id` attribute to `lightning:messageChannel` to access the `publish()` method.

```
<!-- myComponent.cmp -->
<aura:component>
    <lightning:button onclick="{! c.handleClick }"/>
    <lightning:messageChannel type="SampleMessageChannel__c"
        aura:id="sampleMessageChannel"/>
</aura:component>
```

```
// myComponentController.js
({
    handleClick: function(cmp, event, helper) {
        var payload = {
            recordId: "some string",
            recordData: {
                value: "some value"
            }
        };
        cmp.find("sampleMessageChannel").publish(payload);
    }
})
```

In the controller, `handleClick()` contains the `payload` object. This object holds the message that gets sent on the `SampleMessageChannel__c` message channel. Here, the message is a `recordId` with the value "some string" and `recordData`, whose value is the key-value pair `value: "some value"`. Then, the controller finds the `lightning:messageChannel` component referenced in `myComponent.cmp` and calls `publish()` with the payload.

Note: Lightning message service publishes messages to any subscribed component until the destroy phase of the component's lifecycle, even if the component isn't visible. Sometimes when you navigate away from a Lightning page, components are cached and not destroyed. These components still receive messages. For more information, see lifecycle on page 292 and related system events on page 296

Subscribe to a Message Channel

To subscribe to a message channel, create a handler method to run when it receives a message.

Example: The `lmsSubscriberAuraComponent` from the github.com/trailheadapps/lwc-recipes repo shows how to subscribe and unsubscribe from a message channel.

In this example, we define an Aura component called `myNewComponent` that contains the custom message channel, `SampleMessageChannel__c`. The `lightning:messageChannel` component's `onMessage` attribute calls the `handleChanged` method in the client-side controller.

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 `lightning:messageChannel`'s optional parameter, `scope`. Set `scope` to the value `"APPLICATION"`.

```
<lightning:messageChannel type="messageChannel" onMessage="{!listener}"
scope="APPLICATION"/>
```

The component `myNewComponent` detects a new message and updates the display value.

```
<!-- myNewComponent.cmp -->
<aura:component>
    <aura:attribute name="recordValue" type="String"/>
    <lightning:formattedText value="{!v.recordValue}" />
    <lightning:messageChannel type="SampleMessageChannel__c"
```

```
         onMessage="{!c.handleChanged}"/>
</aura:component>
```

```
// myNewComponentController.js
({
    handleChanged: function(cmp, message, helper) {
     // Read the message argument to get the values in the message payload
    if (message != null && message.getParam("recordData") != null) {
        cmp.set("v.recordValue", message.getParam("recordData").value);
    }
  }
})
```

Write the handler in your component's client-side controller. The `handleChanged` method fires when there is a new message. It checks whether there is a payload in the message, and if so, assigns the new data to the `v.recordValue` attribute. The `lightning:formattedText` element updates to display the new value.

Lightning Message Service Limitations

Keep the following in mind when working with Lightning message service.

Supported Experiences

Lightning message service supports only the following 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.

Note: Lightning Message Service doesn't work with Salesforce Tabs + Visualforce sites or with Visualforce pages in Experience Builder sites.

Aura Components That Don't Render Aren't Supported

Lightning message service only supports Aura components that render. You can't use `lightning:messageChannel` in an Aura component that uses the background utility item interface. Similarly, Aura components that use `lightning:messageChannel` can't call Lightning Message Service methods in the `init` lifecycle handler because the component hasn't rendered.

`lightning:messageChannel` Must Be a Child of `aura:component`

In a custom Aura component, `lightning:messageChannel` must be an immediate child of the `aura:component` tag. It can't be nested in an HTML tag or another component.

For example, the following code renders without a problem.

```
<aura:component>
  <lightning:messageChannel type="myMessageChannel__c" />
  <lightning:card>...</lightning:card>
</aura:component>
```

This code throws an error when the Aura component tries to render.

```
<aura:component>
  <lightning:card>
    <lightning:messageChannel type="myMessageChannel__c" />
  </lightning:card>
</aura:component>
```

SEE ALSO:

Invoking Actions on Component Initialization

Component Reference: `lightning:backgroundUtilityItem`

CHAPTER 7 Creating Apps

In this chapter ...

Components are the building blocks of an app. This section shows you a typical workflow to put the pieces together to create a new app.

First, you should decide whether you're creating a component for a standalone app or for Salesforce apps, such as Lightning Experience or Salesforce for Android, iOS, and mobile web. Both components can access your Salesforce data, but only a component created for Lightning Experience or Salesforce for Android, iOS, and mobile web can automatically handle Salesforce events that take advantage of record create and edit pages, among other benefits.

The Quick Start on page 6 walks you through creating components for a standalone app and components for Salesforce for Android, iOS, and mobile web to help you determine which one you need.

App Overview

An app is a special top-level component whose markup is in a `.app` resource.

On a production server, the `.app` resource is the only addressable unit in a browser URL. Access an app using the URL:

`https://`*`MyDomainName`*`.lightning.force.com/<namespace>/<appName>.app`.

SEE ALSO:

aura:application

Supported HTML Tags

Designing App UI

Design your app's UI by including markup in the `.app` resource. Each part of your UI corresponds to a component, which can in turn contain nested components. Compose components to create a sophisticated app.

An app’s markup starts with the `<aura:application>` tag.

Note: Creating a standalone app enables you to host your components outside of Salesforce for Android, iOS, and mobile web or Lightning Experience, such as with Lightning Out or Lightning components in Visualforce pages. To learn more about the `<aura:application>` tag, see aura:application.

Let's look at a `sample.app` file, which starts with the `<aura:application>` tag.

```
<aura:application extends="force:slds">
    <lightning:layout>
        <lightning:layoutItem padding="around-large">
            <h1 class="slds-text-heading_large">Sample App</h1>
        </lightning:layoutItem>
    </lightning:layout>
    <lightning:layout>
        <lightning:layoutItem padding="around-small">
            Sidebar
            <!-- Other component markup here -->
        </lightning:layoutItem>
        <lightning:layoutItem padding="around-small">
            Content
            <!-- Other component markup here -->
        </lightning:layoutItem>
    </lightning:layout>

</aura:application>
```

The `sample.app` file contains HTML tags, such as `<h1>`, as well as components, such as `<lightning:layout>`. We won't go into the details for all the components here but note how simple the markup is. The `<lightning:layoutItem>` component can contain other components or HTML markup.

SEE ALSO:

aura:application

Creating App Templates

An app template bootstraps the loading of the framework and the app. Customize an app's template by creating a component that extends the default `aura:template` template.

A template must have the `isTemplate` system attribute in the `<aura:component>` tag set to `true`. This informs the framework to allow restricted items, such as `<script>` tags, which aren't allowed in regular components.

A component with the `isTemplate` system attribute set to `true` can't be used on a site page. To use a component on a site page, the `isTemplate` system attribute can't be set to `true`.

For example, a sample app has a `np:template` template that extends `aura:template`. `np:template` looks like:

```
<aura:component isTemplate="true" extends="aura:template">
    <aura:set attribute="title" value="My App"/>
    ...
</aura:component>
```

Note how the component extends `aura:template` and sets the `title` attribute using `aura:set`.

The app points at the custom template by setting the `template` system attribute in `<aura:application>`.

```
<aura:application template="np:template">
    ...
</aura:application>
```

A template can only extend a component or another template. A component or an application can't extend a template.

Using the AppCache

AppCache support is deprecated. Browser vendors have deprecated AppCache, so we followed their lead. Remove the `useAppcache` attribute in the `<aura:application>` tag of your standalone apps (`.app` resources) to avoid cross-browser support issues due to deprecation by browser vendors.

If you don't currently set `useAppcache` in an `<aura:application>` tag, you don't have to do anything because the default value of `useAppcache` is `false`.

Note: See an introduction to AppCache for more information.

SEE ALSO:

aura:application

Distributing Applications and Components

As an ISV or Salesforce partner, you can package and distribute applications and components to other Salesforce users and organizations, including those outside your company.

Publish applications and components to and install them from AppExchange.

A managed package ensures that your application and other resources are fully upgradeable. To create and work with managed packages, you must register a namespace prefix. A managed package includes your namespace prefix in the component names and prevents naming conflicts in an installer's organization. After a managed package is released, the application or component names are locked, but the package developer can still edit these attributes.

- API Version
- Description
- Label
- Language
- Markup

IN THIS SECTION:

Apex Class Considerations for Packages
Keep these considerations in mind when you develop Apex classes for packages.

Adding Aura Components to Managed Packages
Add an Aura component to a managed package from a package detail page in Setup.

Deleting Aura Components from Managed Packages
After you've released a managed package, you may decide to refactor the package and delete an Aura component. It's your responsibility to educate your customers about the potential impact from any components you delete. In the Release Notes for your upgraded package, list all custom components you've deleted and notify customers of any necessary actions.

SEE ALSO:

Second-Generation Managed Packaging Developer Guide

First-Generation Managed Packaging Developer Guide

Testing Your Apex Code

Apex Class Considerations for Packages

Keep these considerations in mind when you develop Apex classes for packages.

Test Coverage

Any Apex that is included as part of your definition bundle 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.

Grant User Access for Apex Classes

An authenticated or guest user can access an `@AuraEnabled` Apex method only when the user's profile or an assigned permission set allows access to the Apex class.

- To enable access to a `public` Apex controller that's part of a managed package, a subscriber org must use a permission set. You can't enable access to a `public` Apex controller from a managed package using a user profile.
- To enable access to a `global` Apex controller that's part of a managed package, a subscriber org can use a permission set or a user profile.

Apex Class Usage in Subscriber Orgs

Only methods marked with the `global` access modifier are accessible by Aura components from outside the managed package's namespace. Methods marked with the `public` access modifier are accessible only to Aura components included in the managed package's namespace.

If you declare an Apex method as `global`, you must also declare the Apex class that contains it as `global`.

An Aura component outside the package can access a public Apex method installed from a non-namespaced unlocked package. The Aura component can be installed from another package or created in the org. For accessing Apex methods, a non-namespaced unlocked package is treated the same as an unmanaged package.

SEE ALSO:

Granting User Access for Apex Classes

Apex Server-Side Controller Overview

Apex Developer Guide: Access Modifiers

Adding Aura Components to Managed Packages

Add an Aura component to a managed package from a package detail page in Setup.

When you add an application or component to a package, all definition bundles referenced by the application or component are automatically included, such as other components, events, and interfaces. Custom fields, custom objects, list views, page layouts, and Apex classes referenced by the application or component are also included.

However, when you add a custom object to a package, you must explicitly add the application and other definition bundles that reference that custom object to the package. Other dependencies that you must add to a package explicitly include the following.

- Trusted URLs
- Remote Site Settings

SEE ALSO:

Second-Generation Managed Packaging Developer Guide

First-Generation Managed Packaging Developer Guide

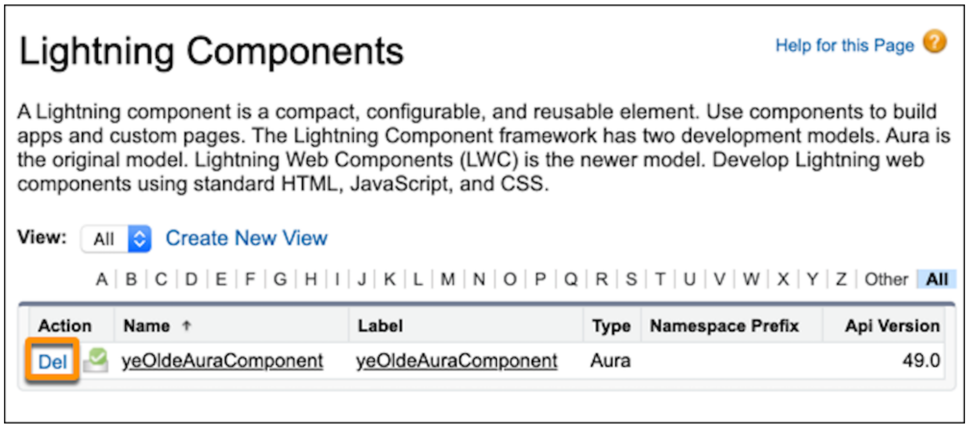
Deleting Aura Components from Managed Packages

After you've released a managed package, you may decide to refactor the package and delete an Aura component. It's your responsibility to educate your customers about the potential impact from any components you delete. In the Release Notes for your upgraded package, list all custom components you've deleted and notify customers of any necessary actions.

Note: To enable component deletion in your packaging org, log a case in the Partner Community.

To delete an Aura component from a managed package:

1. From Setup, enter *`Lightning Components`* in the Quick Find box.
2. Select **Lightning Components**.
3. Click **Del** for the component that you want to delete.



Lightning Components

A Lightning component is a compact, configurable, and reusable element. Use components to build apps and custom pages. The Lightning Component framework has two development models. Aura is the original model. Lightning Web Components (LWC) is the newer model. Develop Lightning web components using standard HTML, JavaScript, and CSS.

View: All Create New View

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | Other | All

Action	Name ↑	Label	Type	Namespace Prefix	Api Version
Del	yeOldeAuraComponent	yeOldeAuraComponent	Aura		49.0

You can delete an Aura component from the Developer Console also.

Note: When a developer removes an Aura component from a package, the component remains in a subscriber's org after they install the upgraded package. The administrator of the subscriber's org can delete the component, if desired. This behavior is the same for an Aura component with a `public` or `global` access value.

The `access` attribute on the `aura:component` tag can be set to `public` or `global` to control whether the component can be used outside of the component's namespace.

We recommend a two-stage process to package developers when you delete an Aura component with `global` access. This process ensures that a global component that you delete from the package has no dependencies on the other items in the package.

1. Stage one: Remove references
   a. Edit the global component that you want to delete to remove all references to other Lightning components.
   b. Upload your new package version.
   c. Push the stage-one upgrade to your subscribers.
2. Stage two: Delete your obsolete component
   a. Delete the global Lightning component from the package.
   b. Optionally, delete other related components and classes.
   c. Upload your new package version.
   d. Push the stage-two upgrade to your subscribers.

SEE ALSO:

Component Access Control

Second-Generation Managed Packaging Developer Guide: Remove Metadata Components from Second-Generation Managed Packages

First-Generation Managed Packaging Developer Guide: Delete Components from First-Generation Managed Packages

CHAPTER 8 Styling Apps

In this chapter ...

An app is a special top-level component whose markup is in a `.app` resource. Just like any other component, you can put CSS in its bundle in a resource called `<appName>.css`.

For example, if the app markup is in `notes.app`, its CSS is in `notes.css`.

When viewed in Salesforce for Android, iOS, and Lightning Experience, the components include styling that matches those visual themes. For example, the `lightning:button` includes the `slds-button_neutral` class to display a neutral style.

Note: Styles added to Lightning components in Salesforce for Android, iOS, and Lightning Experience don't apply to components in standalone apps.

SEE ALSO:

CSS in Components

Using the Salesforce Lightning Design System in Apps

The Salesforce Lightning Design System (SLDS) provides a look and feel that's consistent with Lightning Experience. Use Lightning Design System styles to give your custom stand-alone Lightning applications a UI that is consistent with Salesforce, without having to reverse-engineer our styles.

Your application automatically gets Lightning Design System styles and design tokens if it extends `force:slds`. This method is the easiest way to stay up to date and consistent with Lightning Design System enhancements.

To extend `force:slds`:

```
<aura:application extends="force:slds">
    <!-- customize your application here -->
</aura:application>
```

Using a Static Resource

When you extend `force:slds`, the version of Lightning Design System styles is automatically updated whenever the CSS changes. If you want to use a specific Lightning Design System version, download the version and add it to your org as a static resource.

Note: We recommend extending `force:slds` instead so that you automatically get the latest Lightning Design System styles. If you stick to a specific Lightning Design System version, your app's styles will gradually start to drift from later versions in Lightning Experience or incur the cost of duplicate CSS downloads.

To download a version of Lightning Design System that doesn't exceed the maximum size for a static resource, go to the Lightning Design System downloads page.

Salesforce recommends that you name the Lightning Design System archive static resource using the name format SLDS*###*, where *###* is the Lightning Design System version number (for example, *SLDS252*). This lets you have multiple versions of the Lightning Design System installed, and manage version usage in your components.

To use the static version of the Lightning Design System in a component, include it using `<ltng:require/>`. For example:

```
<aura:component>
    <ltng:require
        styles="{!$Resource.SLDS252 +
            '/styles/salesforce-lightning-design-system.min.css'}" />
</aura:component>
```

SEE ALSO:

Styling with Design Tokens and Styling Hooks

Using External CSS

To reference an external CSS resource, upload it as a static resource and use a `<ltng:require>` tag in your `.cmp` or `.app` markup.

`ltng:require` enables you to load external CSS and JavaScript libraries for your component or app.

Important: You can't load JavaScript resources from a third-party site, even if it's a CSP Trusted Site. To use a JavaScript library from a third-party site, add it to a static resource, and then add the static resource to your component. After the library is loaded from the static resource, you can use it as normal.

Here's an example of using `ltng:require`:

```
<ltng:require styles="{!$Resource.resourceName}" />
```

`resourceName` is the `Name` of the static resource. In a managed package, the resource name must include the package namespace prefix, such as `$Resource.yourNamespace__resourceName`. For a stand-alone static resource, such as an individual graphic or script, you only need the name of the resource. For example, if you uploaded `myScript.js` and set the `Name` to `myScript`, reference it as `$Resource.myScript`. To reference an item within an archive static resource, add the rest of the path to the item using string concatenation.

Here are some considerations for loading styles:

Loading Sets of CSS

Specify a comma-separated list of resources in the `styles` attribute to load a set of CSS.

Note: Due to a quirk in the way `$Resource` is parsed in expressions, use the `join` operator to include multiple `$Resource` references in a single attribute. For example, if you have more than one style sheet to include into a component the `styles` attribute should be something like the following.

```
styles="{!join(',',
    $Resource.myStyles + '/stylesheetOne.css',
    $Resource.myStyles + '/moreStyles.css')}"
```

Loading Order

The styles are loaded in the order that they are listed.

One-Time Loading

The styles load only once, even if they're specified in multiple `<ltng:require>` tags in the same component or across different components.

Encapsulation

To ensure encapsulation and reusability, add the `<ltng:require>` tag to every `.cmp` or `.app` resource that uses the CSS resource.

`ltng:require` also has a `scripts` attribute to load a list of JavaScript libraries. The `afterScriptsLoaded` event enables you to call a controller action after the `scripts` are loaded. It's only triggered by loading of the `scripts` and is never triggered when the CSS in `styles` is loaded.

Styling Components for Lightning Experience or Salesforce for Android, iOS, and mobile web

To prevent styling conflicts in Lightning Experience or Salesforce for Android, iOS, and mobile web, prefix your external CSS with a unique namespace. For example, if you prefix your external CSS declarations with `.myBootstrap`, wrap your component markup with a `<div>` tag that specifies the `myBootstrap` class.

```
<ltng:require styles="{!$Resource.bootstrap}"/>
<div class="myBootstrap">
    <c:myComponent />
    <!-- Other component markup -->
</div>
```

Note: Prefixing your CSS with a unique namespace only applies to external CSS. If you're using CSS within a component bundle, the `.THIS` keyword becomes `.namespaceComponentName` during runtime.

SEE ALSO:

Using External JavaScript Libraries

CSS in Components

$Resource

More Readable Styling Markup with the `join` Expression

Markup can get messy when you specify the class names to apply based on the component attribute values. Try using a `join` expression for easier-to-read markup.

This example sets the class names based on the component attribute values. It's readable, but the spaces between class names are easy to forget.

```
<li class="{! 'calendarEvent ' +
    v.zoomDirection + ' ' +
    (v.past ? 'pastEvent ' : '') +
    (v.zoomed ? 'zoom ' : '') +
    (v.multiDayFragment ? 'multiDayFragment ' : '')}">
    <!-- content here -->
</li>
```

Sometimes, if the markup is not broken into multiple lines, it can hurt your eyes or make you mutter profanities under your breath.

```
<li class="{! 'calendarEvent ' + v.zoomDirection + ' ' + (v.past ? 'pastEvent ' : '') +
(v.zoomed ? 'zoom ' : '') + (v.multiDayFragment ? 'multiDayFragment ' : '')}">
    <!-- content here -->
</li>
```

Try using a `join` expression instead for easier-to-read markup. This example `join` expression sets `' '` as the first argument so that you don't have to specify it for each subsequent argument in the expression.

```
<li
    class="{! join(' ',
        'calendarEvent',
        v.zoomDirection,
        v.past ? 'pastEvent' : '',
        v.zoomed ? 'zoom' : '',
        v.multiDayFragment ? 'multiDayFragment' : ''
    )}">
    <!-- content here -->
</li>
```

You can also use a `join` expression for dynamic styling.

```
<div style="{! join(';',
    'top:' + v.timeOffsetTop + '%',
    'left:' + v.timeOffsetLeft + '%',
    'width:' + v.timeOffsetWidth + '%'
)}">
```

```
    <!-- content here -->
</div>
```

SEE ALSO:

Expression Functions Reference

Tips for CSS in Components

Here are some tips for configuring the CSS for components that you plan to use in Lightning pages, the Lightning App Builder, or the Experience Builder.

Components must be set to 100% width

Because they can be moved to different locations on a Lightning page, components must not have a specific width nor a left or right margin. Components should take up 100% of whatever container they display in. Adding a left or right margin changes the width of a component and can break the layout of the page.

Don't remove HTML elements from the flow of the document

Some CSS rules remove the HTML element from the flow of the document. For example:

```
float: left;
float: right;
position: absolute;
position: fixed;
```

Because they can be moved to different locations on the page as well as used on different pages entirely, components must rely on the normal document flow. Using floats and absolute or fixed positions breaks the layout of the page the component is on. Even if they don't break the layout of the page *you're* looking at, they will break the layout of *some* page the component can be put on.

Child elements shouldn't be styled to be larger than the root element

The Lightning page maintains consistent spacing between components, and can't do that if child elements are larger than the root element.

For example, avoid these patterns:

```
<div style="height: 100px">
  <div style="height: 200px">
    <!--Other markup here-->
  </div>
</div>
```

```
<!--Margin increases the element's effective size-->
<div style="height: 100px">
  <div style="height: 100px margin: 10px">
    <!--Other markup here-->
  </div>
</div>
```

CSS for RTL Languages

When your Language setting in Salesforce is set to a right-to-left (RTL) language, the framework automatically flips property names, such as `left` and `border-left` to `right` and `border-right` respectively. The framework also rearranges certain values like `padding`, `margin`, and `border-radius` so that the `right` and `left` units are swapped.

Flipped CSS Properties

These properties are automatically flipped for RTL languages.

Property	Flipped Property
`left`	`right`
`right`	`left`
`border-left`	`border-right`
`border-left-color`	`border-right-color`
`border-left-style`	`border-right-style`
`border-left-width`	`border-right-width`
`border-right`	`border-left`
`border-right-color`	`border-left-color`
`border-right-style`	`border-left-style`
`border-right-width`	`border-left-width`
`border-top-left-radius`	`border-top-right-radius`
`border-top-right-radius`	`border-top-left-radius`
`border-bottom-left-radius`	`border-bottom-right-radius`
`border-bottom-right-radius`	`border-bottom-left-radius`
`padding-left`	`padding-right`
`padding-right`	`padding-left`
`margin-left`	`margin-right`
`margin-right`	`margin-left`
`nav-left`	`nav-right`
`nav-right`	`nav-left`

Flipped CSS Keywords

These keywords are automatically flipped for RTL languages.

Keyword	Flipped Keyword
`ltr`	`rtl`
`rtl`	`ltr`
`left`	`right`

Keyword	Flipped Keyword
`right`	`left`
`e-resize`	`w-resize`
`w-resize`	`e-resize`
`ne-resize`	`nw-resize`
`nw-resize`	`ne-resize`
`nesw-resize`	`nwse-resize`
`nwse-resize`	`nesw-resize`
`se-resize`	`sw-resize`
`sw-resize`	`se-resize`

Flipped CSS Percentage Values

If the value is a percentage for these properties, the flipped value is set to 100 minus the value.

- `background`
- `background-position`
- `background-position-x`

Flipped Property Arguments

For these properties that can take four values, the second and fourth values are swapped. For example, `property: A B C D` becomes `property: A D C B`.

- `padding`
- `margin`
- `border-color`
- `border-style`
- `border-width`

Flipped `border-radius` Arguments

The arguments for the `border-radius` property are flipped with these patterns.

Arguments	Flipped Arguments
`border-radius: A B`	`border-radius: B A`
`border-radius: A B C`	`border-radius: B A C`
`border-radius: A B C D`	`border-radius: B A D C`

Override Flipping With `@noflip`

To override the automatic flipping, add a `/*@noflip*/` annotation in a comment directly before the property. For example:

```
.THIS.mycontainer {
    /*@noflip*/ direction : rtl;
}
```

Use Conditional CSS

Use the `@if(isRTL)` conditional statement to manually provide the appropriately oriented CSS for each direction.

```
.THIS {
    transform: skew(28deg) translate3d(0, 0 , 0);
}

@if(isRTL) {
    .THIS {
        transform: skew(-28deg) translate3d(0, 0 , 0);
    }
}
```

SEE ALSO:

Salesforce Help: Right-to-Left (RTL) Language Support

Vendor Prefixes

Vendor prefixes, such as `—moz-` and `—webkit-` among many others, are automatically added in Lightning.

You only need to write the unprefixed version, and the framework automatically adds any prefixes that are necessary when generating the CSS output. If you choose to add them, they are used as-is. This enables you to specify alternative values for certain prefixes.

Example: For example, this is an unprefixed version of `border-radius`.

```
.class {
  border-radius: 2px;
}
```

The previous declaration results in the following declarations.

```
.class {
  -webkit-border-radius: 2px;
  -moz-border-radius: 2px;
  border-radius: 2px;
}
```

Styling with Design Tokens and Styling Hooks

Capture the essential values of your visual design into named tokens or global styling hooks. Reuse these values throughout your Lightning components CSS resources. Tokens and styling hooks make it easy to ensure that your design is consistent, and even easier to update your design as it evolves.

Important: Salesforce recommends that you use Styling Hooks instead of design tokens if possible. While existing design tokens still work, styling hooks are the future of customization for Lightning web components and Aura components. See Replace Design Tokens with Styling Hooks.

Design tokens and styling hooks are visual design "atoms" for building a design for your components or apps. Specifically, they're named entities that store visual design attributes: pixel values for margins and spacing, font sizes and families, or hex values for colors. Both design tokens and styling hooks are a terrific way to centralize the low-level values, which you then use to compose the styles that make up the design of your component or app.

IN THIS SECTION:

Tokens Bundles

Tokens are a type of bundle, just like components, events, and interfaces.

Important: Salesforce recommends that you use Styling Hooks instead of design tokens if possible. While existing design tokens still work, styling hooks are the future of customization for Lightning web components and Aura components. See Replace Design Tokens with Styling Hooks.

A tokens bundle contains only one resource, a tokens collection definition.

Resource	Resource Name	Usage
Tokens Collection	`defaultTokens.tokens`	The only required resource in a tokens bundle. Contains markup for one or more tokens. Each tokens bundle contains only one tokens resource.

Note: You can't edit the tokens bundle name or description in the Developer Console after you create it. The bundle's `AuraBundleDefinition` can be modified using the Metadata API.

A tokens collection starts with the `<aura:tokens>` tag. It can only contain `<aura:token>` tags to define tokens.

Tokens collections have restricted support for expressions; see Using Expressions in Tokens. You can't use other markup, renderers, controllers, or anything else in a tokens collection.

SEE ALSO:

Using Expressions in Tokens

Create a Tokens Bundle

Create a tokens bundle in your org using the Developer Console.

Important: Salesforce recommends that you use Styling Hooks instead of design tokens if possible. While existing design tokens still work, styling hooks are the future of customization for Lightning web components and Aura components. See Replace Design Tokens with Styling Hooks.

To create a tokens bundle:

1. In the Developer Console, select **File** > **New** > **Lightning Tokens**.
2. Enter a name for the tokens bundle.

   Your first tokens bundle should be named *`defaultTokens`*. The tokens defined within `defaultTokens` are automatically accessible in your Lightning components. Tokens defined in any other bundle won't be accessible in your components unless you import them into the `defaultTokens` bundle.

You have an empty tokens bundle, ready to edit.

```
<aura:tokens>

</aura:tokens>
```

Note: You can't edit the tokens bundle name or description in the Developer Console after you create it. The bundle's `AuraBundleDefinition` can be modified using the Metadata API. Although you can set a version on a tokens bundle, doing so has no effect.

Defining and Using Tokens

A token is a name-value pair that you specify using the `<aura:token>` component. Define tokens in a tokens bundle, and then use tokens in your components' CSS styles resources.

Important: Salesforce recommends that you use Styling Hooks instead of design tokens if possible. While existing design tokens still work, styling hooks are the future of customization for Lightning web components and Aura components. See Replace Design Tokens with Styling Hooks.

Defining Tokens

Add new tokens as child components of the bundle's `<aura:tokens>` component. For example:

```
<aura:tokens>
    <aura:token name="myBodyTextFontFace"
                value="'Salesforce Sans', Helvetica, Arial, sans-serif"/>
    <aura:token name="myBodyTextFontWeight" value="normal"/>
    <aura:token name="myBackgroundColor" value="#f4f6f9"/>
    <aura:token name="myDefaultMargin" value="6px"/>
</aura:tokens>
```

The only allowed attributes for the `<aura:token>` tag are `name` and `value`.

Using Tokens

Tokens created in the `defaultTokens` bundle are automatically available in components in your namespace. To use a design token, reference it using the `token()` function and the token name in the CSS resource of a component bundle. For example:

```
.THIS p {
    font-family: token(myBodyTextFontFace);
    font-weight: token(myBodyTextFontWeight);
}
```

If you prefer a more concise function name for referencing tokens, you can use the `t()` function instead of `token()`. The two are equivalent. If your token names follow a naming convention or are sufficiently descriptive, the use of the more terse function name won't affect the clarity of your CSS styles.

Using Expressions in Tokens

Tokens support a restricted set of expressions. Use expressions to reuse one token value in another token, or to combine tokens to form a more complex style property.

Important: Salesforce recommends that you use Styling Hooks instead of design tokens if possible. While existing design tokens still work, styling hooks are the future of customization for Lightning web components and Aura components. See Replace Design Tokens with Styling Hooks.

Cross-Referencing Tokens

To reference one token's value in another token's definition, wrap the token to be referenced in standard expression syntax.

In the following example, we reference tokens provided by Salesforce in our custom tokens. Although you can't see the standard tokens directly, imagine that they look something like the following.

```
<!-- force:base tokens (SLDS standard tokens) -->
<aura:tokens>
  ...
  <aura:token name="colorBackground" value="rgb(244, 246, 249)" />
  <aura:token name="fontFamily" value="'Salesforce Sans', Arial, sans-serif" />
  ...
</aura:tokens>
```

With the preceding in mind, you can reference the standard tokens in your custom tokens, as in the following.

```
<!-- defaultTokens.tokens (your tokens) -->
<aura:tokens extends="force:base">
  <aura:token name="mainColor" value="{! colorBackground }" />
  <aura:token name="btnColor" value="{! mainColor }" />
  <aura:token name="myFont" value="{! fontFamily }" />
</aura:tokens>
```

You can only cross-reference tokens defined in the same file or a parent.

Expression syntax in tokens resources is restricted to references to other tokens.

Combining Tokens

To support combining individual token values into more complex CSS style properties, the `token()` function supports string concatenation. For example, if you have the following tokens defined:

```
<!-- defaultTokens.tokens (your tokens) -->
<aura:tokens>
  <aura:token name="defaultHorizonalSpacing" value="12px" />
  <aura:token name="defaultVerticalSpacing" value="6px" />
</aura:tokens>
```

You can combine these two tokens in a CSS style definition. For example:

```
/* myComponent.css */
.THIS div.notification {
  margin: token(defaultVerticalSpacing + ' ' + defaultHorizonalSpacing);
  /* more styles here */
}
```

You can mix tokens with strings as much as necessary to create the right style definition. For example, use `margin: token(defaultVerticalSpacing + ' ' + defaultHorizonalSpacing + ' 3px');` to hard code the bottom spacing in the preceding definition.

The only operator supported within the `token()` function is "+" for string concatenation.

Note: Since Winter '21, we convert Aura tokens to CSS custom properties under the covers. CSS custom properties are a web standard that wasn't supported when we initially created Aura tokens. Concatenating an Aura token with another token that defines a CSS unit isn't supported due to how we convert the Aura tokens. The tokens are statically converted to custom properties and can result in incorrect CSS syntax, which is then discarded by the CSS parser.

For example, don't separate the size and unit into separate tokens.

```
<!-- DO NOT DO THIS! -->
<aura:token name="v24" value="24" />
<aura:token name="px" value="px" />
```

If you concatenate the tokens, the CSS doesn't work as you expect.

```
.THIS { font-size: token(v24+px); }
```

The result is font-size: 24, though you might expect it to be font-size: 24px.

Instead, define a size and unit in one token for this use case.

```
<aura:token name="v24" value="24px" />
```

SEE ALSO:

Defining and Using Tokens

Extending Tokens Bundles

Use the `extends` attribute to extend one tokens bundle from another.

Important: Salesforce recommends that you use Styling Hooks instead of design tokens if possible. While existing design tokens still work, styling hooks are the future of customization for Lightning web components and Aura components. See Replace Design Tokens with Styling Hooks.

To add tokens from one bundle to another, extend the "child" tokens bundle from the "parent" tokens, like this.

```
<aura:tokens extends="yourNamespace:parentTokens">
    <!-- additional tokens here -->
</aura:tokens>
```

Overriding tokens values works mostly as you'd expect: tokens in a child tokens bundle override tokens with the same name from a parent bundle. The exception is if you're using standard tokens. You can't override standard tokens in Lightning Experience or the Salesforce mobile app.

Important: Overriding standard token values is undefined behavior and unsupported. If you create a token with the same name as a standard token, it overrides the standard token's value in some contexts, and has no effect in others. This behavior will change in a future release. Don't use it.

SEE ALSO:

Using Standard Design Tokens

Using Standard Design Tokens

Salesforce exposes a set of "base" tokens that you can access in your component style resources. Use these standard tokens to mimic the look-and-feel of the Salesforce Lightning Design System (SLDS) in your own custom components.

Important: Salesforce recommends that you use Styling Hooks instead of design tokens if possible. While existing design tokens still work, styling hooks are the future of customization for Lightning web components and Aura components. See Replace Design Tokens with Styling Hooks.

To add the standard tokens to your org, extend a tokens bundle from the base tokens, like so.

```
<aura:tokens extends="force:base">
    <!-- your own tokens here -->
</aura:tokens>
```

Once added to `defaultTokens` (or another tokens bundle that `defaultTokens` extends) you can reference tokens from `force:base` just like your own tokens, using the `token()` function and token name. For example:

```
.THIS p {
    font-family: token(fontFamily);
    font-weight: token(fontWeightRegular);
}
```

You can mix-and-match your tokens with the standard tokens. It's a best practice to develop a naming system for your own tokens to make them easily distinguishable from standard tokens. Consider prefixing your token names with "my", or something else easily identifiable.

IN THIS SECTION:

SEE ALSO:

Extending Tokens Bundles

Overriding Standard Tokens (Deprecated)

If you override design tokens for your custom components, replace them with SLDS styling hooks.

Important: Overriding standard tokens is deprecated as of API version 61.0, the Summer '24 release. We recommend that you use Styling Hooks instead. See Replace Design Tokens with Styling Hooks.

SEE ALSO:

Standard Design Tokens—force:base

Standard Design Tokens—`force:base`

The standard tokens available are a subset of the design tokens offered in the Salesforce Lightning Design System (SLDS). The following tokens are available when extending from `force:base`.

Important: Salesforce recommends that you use Styling Hooks instead of design tokens if possible. While existing design tokens still work, styling hooks are the future of customization for Lightning web components and Aura components. See Replace Design Tokens with Styling Hooks.

Available Tokens

Important: The standard token values evolve along with SLDS. Available tokens and their values can change without notice. Token values presented here are for example only.

Token Name	Example Value
`borderWidthThin`	1px
`borderWidthThick`	2px
`spacingXxxSmall`	0.125rem
`spacingXxSmall`	0.25rem
`spacingXSmall`	0.5rem
`spacingSmall`	0.75rem
`spacingMedium`	1rem
`spacingLarge`	1.5rem
`spacingXLarge`	2rem
`varSpacingXxSmall`	0.25rem
`varSpacingXSmall`	0.5rem
`varSpacingSmall`	0.75rem
`varSpacingMedium`	1rem
`varSpacingLarge`	1.5rem
`varSpacingXLarge`	2rem
`varSpacingXxLarge`	3rem
`varSpacingVerticalXxSmall`	0.25rem
`varSpacingVerticalXSmall`	0.5rem
`varSpacingVerticalSmall`	0.75rem
`varSpacingVerticalMedium`	1rem
`varSpacingVerticalLarge`	1.5rem
`varSpacingVerticalXLarge`	2rem
`varSpacingVerticalXxLarge`	3rem
`varSpacingHorizontalXxSmall`	0.25rem
`varSpacingHorizontalXSmall`	0.5rem
`varSpacingHorizontalSmall`	0.75rem
`varSpacingHorizontalMedium`	1rem
`varSpacingHorizontalLarge`	1.5rem
`varSpacingHorizontalXLarge`	2rem
`varSpacingHorizontalXxLarge`	3rem

Token Name	Example Value
`sizeXxSmall`	6rem
`sizeXSmall`	12rem
`sizeSmall`	15rem
`sizeMedium`	20rem
`sizeLarge`	25rem
`sizeXLarge`	40rem
`sizeXxLarge`	60rem
`squareIconUtilitySmall`	1rem
`squareIconUtilityMedium`	1.25rem
`squareIconUtilityLarge`	1.5rem
`squareIconLargeBoundary`	3rem
`squareIconLargeBoundaryAlt`	5rem
`squareIconLargeContent`	2rem
`squareIconMediumBoundary`	2rem
`squareIconMediumBoundaryAlt`	2.25rem
`squareIconMediumContent`	1rem
`squareIconSmallBoundary`	1.5rem
`squareIconSmallContent`	.75rem
`squareIconXSmallBoundary`	1.25rem
`squareIconXSmallContent`	.5rem
`fontWeightLight`	300
`fontWeightRegular`	400
`fontWeightBold`	700
`lineHeightHeading`	1.25
`lineHeightText`	1.375
`lineHeightReset`	1
`lineHeightTab`	2.5rem
`fontFamily`	'Salesforce Sans', Arial, sans-serif
`borderRadiusSmall`	.125rem
`borderRadiusMedium`	.25rem

Token Name	Example Value
`borderRadiusLarge`	.5rem
`borderRadiusPill`	15rem
`borderRadiusCircle`	50%
`colorBorder`	rgb(216, 221, 230)
`colorBorderBrand`	rgb(21, 137, 238)
`colorBorderError`	rgb(194, 57, 52)
`colorBorderSuccess`	rgb(75, 202, 129)
`colorBorderWarning`	rgb(255, 183, 93)
`colorBorderTabSelected`	rgb(0, 112, 210)
`colorBorderSeparator`	rgb(244, 246, 249)
`colorBorderSeparatorAlt`	rgb(216, 221, 230)
`colorBorderSeparatorInverse`	rgb(42, 66, 108)
`colorBorderRowSelected`	rgb(0, 112, 210)
`colorBorderRowSelectedHover`	rgb(21, 137, 238)
`colorBorderButtonBrand`	rgb(0, 112, 210)
`colorBorderButtonBrandDisabled`	rgba(0, 0, 0, 0)
`colorBorderButtonDefault`	rgb(216, 221, 230)
`colorBorderButtonInverseDisabled`	rgba(255, 255, 255, 0.15)
`colorBorderInput`	rgb(216, 221, 230)
`colorBorderInputActive`	rgb(21, 137, 238)
`colorBorderInputDisabled`	rgb(168, 183, 199)
`colorBorderInputCheckboxSelectedCheckmark`	rgb(255, 255, 255)
`colorBackground`	rgb(244, 246, 249)
`colorBackgroundAlt`	rgb(255, 255, 255)
`colorBackgroundAltInverse`	rgb(22, 50, 92)
`colorBackgroundRowHover`	rgb(244, 246, 249)
`colorBackgroundRowActive`	rgb(238, 241, 246)
`colorBackgroundRowSelected`	rgb(240, 248, 252)
`colorBackgroundRowNew`	rgb(217, 255, 223)
`colorBackgroundInverse`	rgb(6, 28, 63)

Token Name	Example Value
`colorBackgroundBrowser`	rgb(84, 105, 141)
`colorBackgroundChromeMobile`	rgb(0, 112, 210)
`colorBackgroundChromeDesktop`	rgb(255, 255, 255)
`colorBackgroundHighlight`	rgb(250, 255, 189)
`colorBackgroundModal`	rgb(255, 255, 255)
`colorBackgroundModalBrand`	rgb(0, 112, 210)
`colorBackgroundNotificationBadge`	rgb(194, 57, 52)
`colorBackgroundNotificationBadgeHover`	rgb(0, 95, 178)
`colorBackgroundNotificationBadgeFocus`	rgb(0, 95, 178)
`colorBackgroundNotificationBadgeActive`	rgb(0, 57, 107)
`colorBackgroundNotificationNew`	rgb(240, 248, 252)
`colorBackgroundPayload`	rgb(244, 246, 249)
`colorBackgroundShade`	rgb(224, 229, 238)
`colorBackgroundStencil`	rgb(238, 241, 246)
`colorBackgroundStencilAlt`	rgb(224, 229, 238)
`colorBackgroundScrollbar`	rgb(224, 229, 238)
`colorBackgroundScrollbarTrack`	rgb(168, 183, 199)
`colorBrand`	rgb(21, 137, 238)
`colorBrandDark`	rgb(0, 112, 210)
`colorBackgroundModalButton`	rgba(0, 0, 0, 0.07)
`colorBackgroundModalButtonActive`	rgba(0, 0, 0, 0.16)
`colorBackgroundInput`	rgb(255, 255, 255)
`colorBackgroundInputActive`	rgb(255, 255, 255)
`colorBackgroundInputCheckbox`	rgb(255, 255, 255)
`colorBackgroundInputCheckboxDisabled`	rgb(216, 221, 230)
`colorBackgroundInputCheckboxSelected`	rgb(21, 137, 238)
`colorBackgroundInputDisabled`	rgb(224, 229, 238)
`colorBackgroundInputError`	rgb(255, 221, 225)
`colorBackgroundPill`	rgb(255, 255, 255)
`colorBackgroundToast`	rgba(84, 105, 141, 0.95)

Token Name	Example Value
`colorBackgroundToastSuccess`	rgb(4, 132, 75)
`colorBackgroundToastError`	rgba(194, 57, 52, 0.95)
`shadowDrag`	0 2px 4px 0 rgba(0, 0, 0, 0.40)
`shadowDropDown`	0 2px 3px 0 rgba(0, 0, 0, 0.16)
`shadowHeader`	0 2px 4px rgba(0, 0, 0, 0.07)
`shadowButtonFocus`	0 0 3px #0070D2
`shadowButtonFocusInverse`	0 0 3px #E0E5EE
`colorTextActionLabel`	rgb(84, 105, 141)
`colorTextActionLabelActive`	rgb(22, 50, 92)
`colorTextBrand`	rgb(21, 137, 238)
`colorTextBrowser`	rgb(255, 255, 255)
`colorTextBrowserActive`	rgba(0, 0, 0, 0.4)
`colorTextDefault`	rgb(22, 50, 92)
`colorTextError`	rgb(194, 57, 52)
`colorTextInputDisabled`	rgb(84, 105, 141)
`colorTextInputFocusInverse`	rgb(22, 50, 92)
`colorTextInputIcon`	rgb(159, 170, 181)
`colorTextInverse`	rgb(255, 255, 255)
`colorTextInverseWeak`	rgb(159, 170, 181)
`colorTextInverseActive`	rgb(94, 180, 255)
`colorTextInverseHover`	rgb(159, 170, 181)
`colorTextLink`	rgb(0, 112, 210)
`colorTextLinkActive`	rgb(0, 57, 107)
`colorTextLinkDisabled`	rgb(22, 50, 92)
`colorTextLinkFocus`	rgb(0, 95, 178)
`colorTextLinkHover`	rgb(0, 95, 178)
`colorTextLinkInverse`	rgb(255, 255, 255)
`colorTextLinkInverseHover`	rgba(255, 255, 255, 0.75)
`colorTextLinkInverseActive`	rgba(255, 255, 255, 0.5)
`colorTextLinkInverseDisabled`	rgba(255, 255, 255, 0.15)

Token Name	Example Value
`colorTextModal`	rgb(255, 255, 255)
`colorTextModalButton`	rgb(84, 105, 141)
`colorTextStageLeft`	rgb(224, 229, 238)
`colorTextTabLabel`	rgb(22, 50, 92)
`colorTextTabLabelSelected`	rgb(0, 112, 210)
`colorTextTabLabelHover`	rgb(0, 95, 178)
`colorTextTabLabelFocus`	rgb(0, 95, 178)
`colorTextTabLabelActive`	rgb(0, 57, 107)
`colorTextTabLabelDisabled`	rgb(224, 229, 238)
`colorTextToast`	rgb(224, 229, 238)
`colorTextWeak`	rgb(84, 105, 141)
`colorTextIconBrand`	rgb(0, 112, 210)
`colorTextButtonBrand`	rgb(255, 255, 255)
`colorTextButtonBrandHover`	rgb(255, 255, 255)
`colorTextButtonBrandActive`	rgb(255, 255, 255)
`colorTextButtonBrandDisabled`	rgb(255, 255, 255)
`colorTextButtonDefault`	rgb(0, 112, 210)
`colorTextButtonDefaultHover`	rgb(0, 112, 210)
`colorTextButtonDefaultActive`	rgb(0, 112, 210)
`colorTextButtonDefaultDisabled`	rgb(216, 221, 230)
`colorTextButtonDefaultHint`	rgb(159, 170, 181)
`colorTextButtonInverse`	rgb(224, 229, 238)
`colorTextButtonInverseDisabled`	rgba(255, 255, 255, 0.15)
`colorTextIconDefault`	rgb(84, 105, 141)
`colorTextIconDefaultHint`	rgb(159, 170, 181)
`colorTextIconDefaultHover`	rgb(0, 112, 210)
`colorTextIconDefaultActive`	rgb(0, 57, 107)
`colorTextIconDefaultDisabled`	rgb(216, 221, 230)
`colorTextIconInverse`	rgb(255, 255, 255)
`colorTextIconInverseHover`	rgb(255, 255, 255)

Token Name	Example Value
`colorTextIconInverseActive`	rgb(255, 255, 255)
`colorTextIconInverseDisabled`	rgba(255, 255, 255, 0.15)
`colorTextLabel`	rgb(84, 105, 141)
`colorTextPlaceholder`	rgb(84, 105, 141)
`colorTextPlaceholderInverse`	rgb(224, 229, 238)
`colorTextRequired`	rgb(194, 57, 52)
`colorTextPill`	rgb(0, 112, 210)
`durationInstantly`	0s
`durationImmediately`	0.05s
`durationQuickly`	0.1s
`durationPromptly`	0.2s
`durationSlowly`	0.4s
`durationPaused`	3.2s
`colorBackgroundButtonBrand`	rgb(0, 112, 210)
`colorBackgroundButtonBrandActive`	rgb(0, 57, 107)
`colorBackgroundButtonBrandHover`	rgb(0, 95, 178)
`colorBackgroundButtonBrandDisabled`	rgb(224, 229, 238)
`colorBackgroundButtonDefault`	rgb(255, 255, 255)
`colorBackgroundButtonDefaultHover`	rgb(244, 246, 249)
`colorBackgroundButtonDefaultFocus`	rgb(244, 246, 249)
`colorBackgroundButtonDefaultActive`	rgb(238, 241, 246)
`colorBackgroundButtonDefaultDisabled`	rgb(255, 255, 255)
`colorBackgroundButtonIcon`	rgba(0, 0, 0, 0)
`colorBackgroundButtonIconHover`	rgb(244, 246, 249)
`colorBackgroundButtonIconFocus`	rgb(244, 246, 249)
`colorBackgroundButtonIconActive`	rgb(238, 241, 246)
`colorBackgroundButtonIconDisabled`	rgb(255, 255, 255)
`colorBackgroundButtonInverse`	rgba(0, 0, 0, 0)
`colorBackgroundButtonInverseActive`	rgba(0, 0, 0, 0.24)
`colorBackgroundButtonInverseDisabled`	rgba(0, 0, 0, 0)

Token Name	Example Value
`lineHeightButton`	1.875rem
`lineHeightButtonSmall`	1.75rem
`colorBackgroundAnchor`	rgb(244, 246, 249)

For a complete list of the design tokens available in the SLDS, see Design Tokens on the Lightning Design System site.

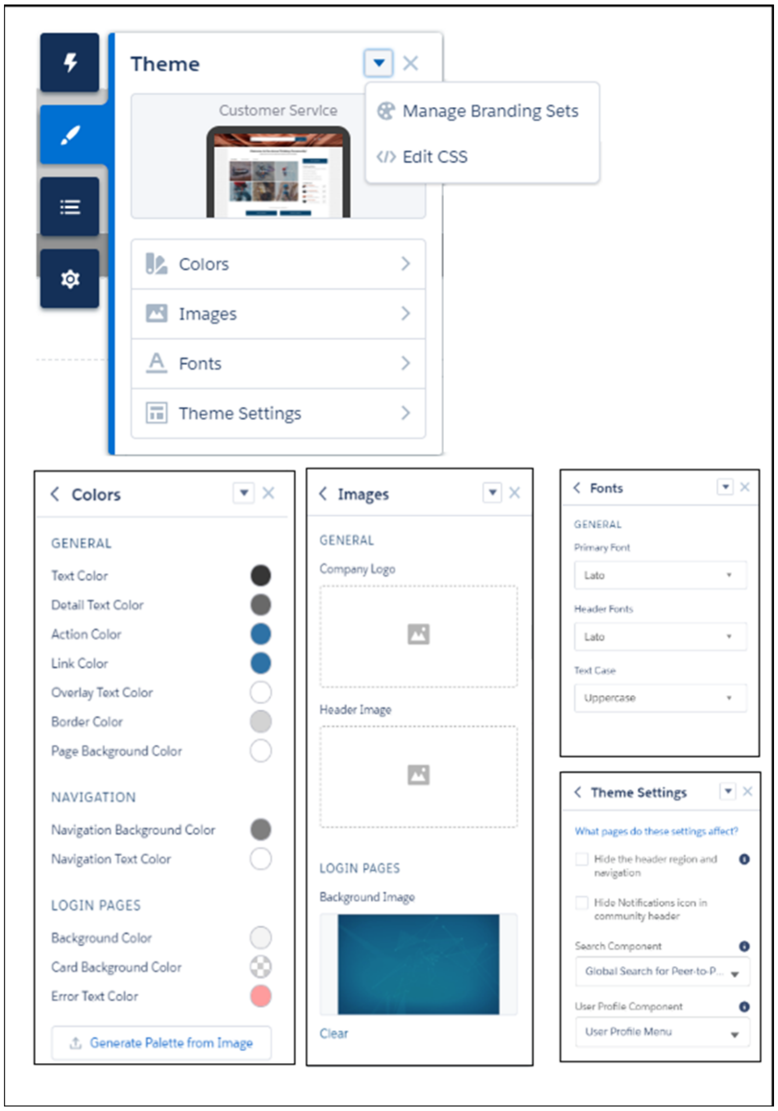
SEE ALSO:

Extending Tokens Bundles

Standard Design Tokens for Experience Builder Sites

Use a subset of the standard design tokens to make your components compatible with the Theme panel in Experience Builder. The Theme panel enables administrators to quickly style an entire site using these properties. Each property in the Theme panel maps to one or more standard design tokens. When an administrator updates a property in the Theme panel, the system automatically updates any Lightning components that use the tokens associated with that branding property.

Important: Salesforce recommends that you use Styling Hooks instead of design tokens if possible. While existing design tokens still work, styling hooks are the future of customization for Lightning web components and Aura components. See Replace Design Tokens with Styling Hooks.



Primary Font: Lato
Header Fonts: Lato
Text Case: Uppercase
Hide the header region and navigation
Hide Notifications icon in community header
Search Component: Global Search for Peer-to-P...
User Profile Component: User Profile Menu

Available Tokens for Experience Builder Sites

For Experience Builder sites, these standard tokens are available when extending from `forceCommunity:base`.

Important: The standard token values evolve along with SLDS. Available tokens and their values can change without notice.

Important: Design tokens are not available for navigation branding properties. To add branding to navigation properties, override the navigation bar within the custom components. See CSS Overrides Migration for the Navigation Menu.

These Branding panel properties...	...map to these standard design tokens
Text Color	`colorTextDefault`
Detail Text Color	• `colorTextActionLabel` • `colorTextLabel` • `colorTextPlaceholder`

These Branding panel properties...	...map to these standard design tokens
	• `colorTextWeak`
Action Color	• `colorBackgroundButtonBrand` • `colorBorderBrand` • `colorBorderButtonBrand` • `colorBrand` • `colorTextBrand` • `colorTextTabLabelSelected` • `colorTextActionLabelActive` Note: As of Summer '18 `colorBackgroundHighlight` is no longer mapped to Action Color.
Link Color	`colorTextLink`
Overlay Text Color	• `colorTextButtonBrand` • `colorTextButtonBrandHover` • `colorTextInverse`
Border Color	• `colorBorder` • `colorBorderButtonDefault` • `colorBorderInput` • `colorBorderSeparatorAlt`
Company Logo	`brandLogoImage`
Header Image	`headerImageUrl`
Login Pages Background Image	`LoginBackgroundImage`
Primary Font	`fontFamily`
Text Case	`textTransform`

In addition, the following standard tokens are available for derived theme properties in the template. You can indirectly access derived properties when you update the properties in the Theme panel. For example, if you change the Action Color property in the Theme panel, the system automatically recalculates the Action Color Darker value based on the new value.

These derived branding properties...	...map to these standard design tokens
Action Color Darker (Derived from Action Color)	• `colorBackgroundButtonBrandActive` • `colorBackgroundButtonBrandHover`
Hover Color (Derived from Action Color)	• `colorBackgroundButtonDefaultHover` • `colorBackgroundRowHover`

These derived branding properties...	...map to these standard design tokens
	• `colorBackgroundRowSelected` • `colorBackgroundShade`
Link Color Darker (Derived from Link Color)	• `colorTextLinkActive` • `colorTextLinkHover`

For a complete list of the design tokens available in the SLDS, see Design Tokens on the Lightning Design System site.

SEE ALSO:

Configure Components for Experience Builder

Replace Design Tokens with Styling Hooks

If you use design tokens to customize the styling of your Aura components, use SLDS global styling hooks instead. Custom components that use design tokens still work, but they no longer receive updates after LWC API version 61.0. By using styling hooks, you can cleanly adopt future product innovations and updated web accessibility standards.

Replace Design Tokens with Styling Hooks

Directly replace design tokens with `--slds` styling hooks in your Aura CSS file.

```
/* Aura Custom Component CSS */
.THIS .my-custom-container {
  background-color: var(--slds-g-color-surface-container-1);
}
```

Most of the customization options provided by design tokens are available with SLDS global styling hooks. For a full list of global styling hooks, see the Global Styling Hooks Reference.

Styling if Styling Hooks are Unavailable

Only Lightning Experience supports the latest global styling hooks. In containers such as Experience Cloud sites, newer styling hooks, such as the `--slds-g-color-*` styling hooks, aren't available. To accommodate containers that can't access these styling hooks, include an `--lwc` custom property as a fallback. Use this solution only if the component is expected to run in a container that doesn't support styling hooks.

Convert a design token to an `--lwc` property by adding `--lwc` as a prefix to the design token name. For example, instead of `t(colorTextBrand)`, use `var(--lwc-colorTextBrand)`.

Example: This example shows an Aura CSS file for a custom component that uses a design token to override the component's background color.

```
/* Aura CSS using an Aura token to override*/
.THIS .my-custom-container {
  background-color: t(cardColorBackground);
}
```

The best replacement is the new `--slds-g-color-*` styling hooks. However, in this case the container can't access newer styling hooks. So this example replaces the design token by referencing a global color styling hook and also an `--lwc` property that's derived from the original design token.

```
/* Aura Custom Component CSS */
.THIS .my-custom-container {
  background-color: var(--slds-g-color-surface-container-1, --lwc-cardColorBackground);
}
```

SEE ALSO:

Salesforce Lightning Design System: Styling Hooks

CHAPTER 9 Developing Secure Code

Aura components have a client-side security architecture that helps protect your custom components by automatically blocking or modifying any insecure behavior of APIs. This layer prevents components from accessing data that belongs to platform code or components from other namespaces without explicit permission.

To learn how to build components that work with Lightning Web Security (LWS) or the legacy architecture Lightning Locker, see the Security for Lightning Components guide.

The framework also uses JavaScript strict mode to turn on native security features in the browser, and Content Security Policy (CSP) rules to control the source of content that can be loaded on a page.

SEE ALSO:

Security for Lightning Components: Compare Lightning Web Security to Lightning Locker

CHAPTER 10 Using JavaScript

In this chapter ...

Use JavaScript for client-side code. The `$A` namespace is the entry point for using the framework in JavaScript code.

For all the methods available in `$A`, see the JavaScript API.

A component bundle can contain JavaScript code in a client-side controller, helper, or renderer. Client-side controllers are the most commonly used of these JavaScript resources.

Expressions in JavaScript Code

In JavaScript, use string syntax to evaluate an expression. For example, this expression retrieves the `label` attribute in a component.

```
var theLabel = cmp.get("v.label");
```

Note: Only use the `{! }` expression syntax in markup in `.app` or `.cmp` resources.

SEE ALSO:

Handling Events with Client-Side Controllers

- Calling Component Methods
- Dynamically Adding Event Handlers To a Component
- Dynamically Showing or Hiding Markup
- Adding and Removing Styles
- Which Button Was Pressed?
- Formatting Dates in JavaScript
- Using JavaScript Promises
- Making API Calls from Components
- Control Access to Browser Features
- Manage Trusted URLs

Supported JavaScript

The Aura Components programming model supports ES5 syntax and ES6 Promises.

For the most reliable experience, use ES5 to develop Aura components because the pipeline from authoring to serialization to execution was built for ES5. Promises from ES6 are also available. Using any other syntax or feature is not supported.

This developer guide explains how to develop Aura components and documents the JavaScript usage that's unique to the Aura Components programming model.

If you want to use ES6 or later for development, use the Lightning Web Components programming model, which has been architected for modern JavaScript development.

SEE ALSO:

Browser Support for Aura Components

Invoking Actions on Component Initialization

Use the `init` event to initialize a component or fire an event after component construction but before rendering.

Note: The `init` event is fired only once per lifecycle of the component. The `init` event doesn't get fired if the component is served from cache. To execute JavaScript code every time a component is rendered, use the `render` event instead.

Let's look at an example.

```
<!--initCmp.cmp-->
<aura:component>
    <aura:attribute name="setMeOnInit" type="String" default="default value" />
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <p>This value is set in the controller after the component initializes and before
rendering.</p>
    <p><b>{!v.setMeOnInit}</b></p>

</aura:component>
```

The magic happens in this line.

```
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
```

This code registers an `init` event handler for the component. `init` is a predefined event sent to every component. Setting `value="{!this}"` marks this as a value event. You should always use this setting for an `init` event.

After the component is initialized, the `doInit` action is called in the component's controller.

```
// initCmp.js
({
    doInit: function(cmp) {
        // Set the attribute value.
        // You could also fire an event here instead.
        cmp.set("v.setMeOnInit", "controller init magic!");
    }
})
```

The `doInit` action sets an attribute value, but it could do something more interesting, such as firing an event.

If a component is contained in another component or app, the inner component is initialized first.

SEE ALSO:

Handling Events with Client-Side Controllers

Handle the render Event

Component Attributes

Detecting Data Changes with Change Handlers

Sharing JavaScript Code in a Component Bundle

Put functions that you want to reuse in the component's helper. Helper functions also enable specialization of tasks, such as processing data and queueing server-side actions. Helper functions are local to a component, improve code reuse, and move the heavy lifting of JavaScript logic away from the client-side controller, where possible.

A helper function can be called from any JavaScript code in a component's bundle, such as from a client-side controller or renderer.

Helper functions are similar to client-side controller functions in shape, surrounded by parentheses and curly braces to denote a JavaScript object in object-literal notation containing a map of name-value pairs. A helper function can pass in any arguments required by the function, such as the component it belongs to, a callback, or any other objects.

```
({
    helperMethod1 : function() {
        // logic here
    },

    helperMethod2 : function(component) {
        // logic here
        this.helperMethod3(var1, var2);
    },

     helperMethod3 : function(var1, var2) {
         // do something with var1 and var2 here
    }
})
```

To call another function in the same helper, use the syntax: `this.`*`methodName`*, where `this` is a reference to the helper itself. For example, `helperMethod2` calls `helperMethod3` with this code.

```
this.helperMethod3(var1, var2);
```

Creating a Helper

A helper resource is part of the component bundle and is auto-wired via the naming convention, *`<componentName>`*`Helper.js`.

To create a helper using the Developer Console, click **HELPER** in the sidebar of the component. This helper file is valid for the scope of the component to which it's auto-wired.

Using a Helper in a Controller

Add a `helper` argument to a controller function to enable the function to use the helper. Specify `(component, event, helper)` in the controller. These are standard parameters and you don't have to access them in the function.

This controller code calls an `updateItem` helper function.

```
/* controller */
({
    newItemEvent: function(component, event, helper) {
        helper.updateItem(component, event.getParam("item"));
    }
})
```

Here's the helper that contains the `updateItem` function called by the controller.

```
/* helper */
({
    updateItem : function(component, item, callback) {
        // Update the items via a server-side action
        var action = component.get("c.saveItem");
        action.setParams({"item" : item});
        // Set any optional callback and enqueue the action
        if (callback) {
            action.setCallback(this, callback);
        }
        $A.enqueueAction(action);
    }
})
```

The `updateItem` function accepts three parameters.

1. `component`—The component to which the helper belongs.
2. `item`—An item that's set as an `item` parameter for the `saveItem` Apex action.
3. `callback`—An optional callback to call after the `saveItem` Apex action returns. In our example, the `newItemEvent` controller method passes in only two arguments so there's no callback.

Using a Helper in a Renderer

Add a helper argument to a renderer function to enable the function to use the helper. In the renderer, specify `(component, helper)` as parameters in a function signature to enable the function to access the component's helper. These are standard parameters and you don't have to access them in the function. The following code shows an example on how you can override the `afterRender()` function in the renderer and call `open` in the helper method.

detailsRenderer.js

```
({
    afterRender : function(component, helper){
        helper.open(component, null, "new");
    }
})
```

detailsHelper.js

```
({
    open : function(component, note, mode, sort){
        if(mode === "new") {
            //do something
        }
        // do something else, such as firing an event
    }
})
```

SEE ALSO:

Create a Custom Renderer

Component Bundles

Handling Events with Client-Side Controllers

Sharing JavaScript Code Across Components

You can build simple Lightning components that are entirely self-contained. However, if you build more complex applications, you probably want to share code, or even client-side data, between components.

The `<ltng:require>` tag enables you to load external JavaScript libraries after you upload them as static resources. You can also use `<ltng:require>` to import your own JavaScript libraries of utility methods.

Let's look at a simple counter library that provides a `getValue()` method, which returns the current value of the counter, and an `increment()` method, which increments the value of that counter.

Create the JavaScript Library

1. In the Developer Console, click **File** > **New** > **Static Resource**.
2. Enter *counter* in the `Name` field.
3. Select *text/javascript* in the `MIME Type` field.
4. Click **Submit**.
5. Enter this code and click **File** > **Save**.

```
window._counter = (function() {
    var value = 0; // private

    return { //public API
        increment: function() {
            value = value + 1;
            return value;
        },

        getValue: function() {
            return value;
        }
    };
}());
```

This code uses the JavaScript module pattern. Using this closure-based pattern, the `value` variable remains private to your library. Components using the library can't access `value` directly.

The most important line of the code to note is:

```
window._counter = (function() {
```

You must attach `_counter` to the `window` object as a requirement of JavaScript strict mode, which is implicitly enabled in Lightning Locker. Even though `window._counter` looks like a global declaration, `_counter` is attached to the Lightning Locker secure window object and therefore is a namespace variable, not a global variable.

If you use `_counter` instead of `window._counter`, `_counter` isn't available. When you try to access it, you get an error similar to:

```
Action failed: ... [_counter is not defined]
```

Use the JavaScript Library

Let's use the library in a `MyCounter` component that has a simple UI to exercise the `counter` methods.

```
<!--c:MyCounter-->
<aura:component access="global">
    <ltng:require scripts="{!$Resource.counter}"
                  afterScriptsLoaded="{!c.getValue}"/>
    <aura:attribute name="value" type="Integer"/>

    <h1>MyCounter</h1>
    <p>{!v.value}</p>
    <lightning:button label="Get Value" onclick="{!c.getValue}"/>
    <lightning:button label="Increment" onclick="{!c.increment}"/>
</aura:component>
```

The `<ltng:require>` tag loads the counter library and calls the `getValue` action in the component's client-side controller after the library is loaded.

Here's the client-side controller.

```
/* MyCounterController.js */
({
    getValue : function(component, event, helper) {
        component.set("v.value", _counter.getValue());
    },

    increment : function(component, event, helper) {
        component.set("v.value", _counter.increment());
    }
})
```

You can access properties of the `window` object without having to type the `window.` prefix. Therefore, you can use `_counter.getValue()` as shorthand for `window._counter.getValue()`.

Click the buttons to get the value or increment it.

Our counter library shares the counter value between any components that use the library. If you need each component to have a separate counter, you could modify the counter implementation. To see the per-component code and for more details, see this blog post about *Modularizing Code in Lightning Components*.

SEE ALSO:

Using External JavaScript Libraries

Security for Lightning Components: JavaScript Strict Mode Enforcement

Using External JavaScript Libraries

To reference a JavaScript library, upload it as a static resource and use a `<ltng:require>` tag in your `.cmp` or `.app` markup.

Note: Before you use a third-party JavaScript library, we recommend that you check AppExchange for components or apps from Salesforce partners that match your requirements. Alternatively, check if a base component provides your desired functionality.

The framework's content security policy mandates that external JavaScript libraries must be uploaded to Salesforce static resources.

You can't use a `<script>` tag in a component. This restriction mitigates the risk of cross-site scripting attacks. You can add a `<script>` tag to an application's template, which is a special type of component that extends `aura:template`.

Note: Only third-party JavaScript libraries that are loaded via `ltng:require` are supported. Documentation and examples that demonstrate using a third-party JavaScript library don't constitute an endorsement of that library. We recommend that you check the third-party JavaScript library documentation for usage information.

Here's an example of using `ltng:require`.

```
<ltng:require scripts="{!$Resource.resourceName}"
    afterScriptsLoaded="{!c.afterScriptsLoaded}" />
```

`resourceName` is the `Name` of the static resource. In a managed package, the resource name must include the package namespace prefix, such as `$Resource.yourNamespace__resourceName`. For a stand-alone static resource, such as an individual graphic or script, you only need the name of the resource. For example, if you uploaded `myScript.js` and set the `Name` to `myScript`, reference it as `$Resource.myScript`. To reference an item within an archive static resource, add the rest of the path to the item using string concatenation.

The `afterScriptsLoaded` action in the client-side controller is called after the scripts are loaded and the component is rendered. Don't use the `init` event to access scripts loaded by `ltng:require`. These scripts load asynchronously and are most likely not available when the `init` event handler is called.

Here are some considerations for loading scripts:

Loading Sets of Scripts

Specify a comma-separated list of resources in the `scripts` attribute to load a set of resources.

Note: Due to a quirk in the way `$Resource` is parsed in expressions, use the `join` operator to include multiple `$Resource` references in a single attribute. For example, if you have more than one JavaScript library to include into a component the `scripts` attribute should be something like the following.

```
scripts="{!join(',',
    $Resource.jsLibraries + '/jsLibOne.js',
    $Resource.jsLibraries + '/jsLibTwo.js')}"
```

Loading Order

The scripts are loaded in the order that they are listed.

One-Time Loading

Scripts load only once, even if they're specified in multiple `<ltng:require>` tags in the same component or across different components.

Parallel Loading

Use separate `<ltng:require>` tags for parallel loading if you have multiple sets of scripts that are not dependent on each other.

Encapsulation

To ensure encapsulation and reusability, add the `<ltng:require>` tag to every `.cmp` or `.app` resource that uses the JavaScript library.

`ltng:require` also has a `styles` attribute to load a list of CSS resources. You can set the `scripts` and `styles` attributes in one `<ltng:require>` tag.

Using a Client-Side Controller with External JavaScript Libraries

If you're using an external library to work with your HTML elements after rendering, use `afterScriptsLoaded` to wire up a client-side controller. The following example sets up a chart using the `Chart.js` library, which is uploaded as a static resource.

```
<ltng:require scripts="{!$Resource.chart}"
              afterScriptsLoaded="{!c.setup}"/>
<canvas aura:id="chart" id="myChart" width="400" height="400"/>
```

The component's client-side controller sets up the chart after component initialization and rendering.

```
setup : function(component, event, helper) {
    var data = {
        labels: ["January", "February", "March"],
        datasets: [{
            data: [65, 59, 80, 81, 56, 55, 40]
        }]
    };
    var el = component.find("chart").getElement();
    var ctx = el.getContext("2d");
    var myNewChart = new Chart(ctx).Line(data);
}
```

Troubleshooting Errors from `ltng:require`

Let's say your component references a custom JavaScript library with `ltng:require`. When you try to load the component, a modal dialog interrupts and displays information about an error.

For example, the dialog could show a message like the following.

```
Custom Script Eval error in 'ltng:require' [SecureDOMEvent: [object Event] {key:
{namespace":"c"}}]
```

The dialog could also include a stack trace. If it doesn't, check the browser's JavaScript console for more information. If the component didn't load, the console doesn't show much and the problem is likely in the library you referenced.

Use the Locker Console to evaluate the JavaScript from the library to see if it's affected by Locker restrictions.

If `ltng:require` encounters errors in your script, you see an error in the JavaScript console that includes details about the problem. The JavaScript console could show a message such as the following.

```
WARNING: Failed to load script at
/resource/156768268766/MyHeader/static/myLib.js:
Cannot assign to read only property 'someProp' of object '[object Object]'
```

This also indicates the problem is in the static resource, `myLib.js` in this case. If the Locker Console gives you the same message when you evaluate the JavaScript from `myLib.js`, this confirms that the script is attempting to perform an action that is not allowed by Locker.

SEE ALSO:

Salesforce Help: Static Resources

$Resource

Using External CSS

Component Library

Security for Lightning Components: Content Security Policy Overview

Creating App Templates

Dynamically Creating Components

Create a component dynamically in your client-side JavaScript code by using the `$A.createComponent()` method. To create multiple components, use `$A.createComponents()`.

Note: Use `$A.createComponent()` instead of the deprecated `$A.newCmp()` and `$A.newCmpAsync()` methods.

Client-Side Versus Server-Side Component Creation

The `$A.createComponent()` and `$A.createComponents()` methods support both client-side (synchronous) and server-side (asynchronous) component creation. For performance and other reasons, client-side creation is preferred.

To use `$A.createComponent()`, we need the component definition. If we don't have the definition already on the client, the framework makes a server trip to get it. You can avoid this server trip by adding an `<aura:dependency>` tag for the component you're creating in the markup of the component that calls `$A.createComponent()`. The tag ensures that the component definition is always available on the client. The tradeoff is that the definition is always downloaded instead of only when it's needed. This performance tradeoff decision depends on your use case.

If no server-side dependencies are found, the methods are executed synchronously on the client-side. The top-level component determines whether a server request is necessary for component creation. A component with server-side dependencies must be created on the server. Server-side dependencies include component definitions or dynamically loaded labels that aren't already on the client, and other elements that can't be predetermined by static markup analysis.

Note: A server-side controller isn't a server-side dependency for component creation because controller actions are only called after the component has been created.

A single call to `createComponent()` or `createComponents()` can result in many components being created. The call creates the requested component and all its child components. In addition to performance considerations, server-side component creation has a limit of 10,000 components that can be created in a single request. If you hit this limit, explicitly declare component dependencies with the `<aura:dependency>` tag or otherwise pre-load dependent elements. The components are then created on the client side instead.

There's no limit on component creation on the client side.

Note: Creating components where the top-level components don't have server dependencies but nested inner components do have dependencies isn't currently supported.

Syntax

The syntax is:

```
$A.createComponent(String type, Object attributes, function callback)
```

1. `type`—The type of component to create; for example, `"lightning:button"`.
2. `attributes`—A map of attributes for the component, including the local Id (`aura:id`).
3. `callback(cmp, status, errorMessage)`—The callback to invoke after the component is created.

   Tip: Component creation is asynchronous if it requires a server trip. Follow good asynchronous practices, such as only using the new component in the callback.

   The callback has three parameters.

   a. `cmp`—The component that was created. This parameter enables you to do something with the new component, such as add it to the body of the component that creates it. If there's an error, `cmp` is `null`.

   b. `status`—The status of the call. The possible values are `SUCCESS`, `INCOMPLETE`, or `ERROR`. Always check that the status is `SUCCESS` before you try to use the component.

   c. `errorMessage`—The error message if the status is `ERROR`.

Example

Let's add a dynamically created button to this sample component.

```
<!--c:createComponent-->
<aura:component>
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <p>Dynamically created button</p>
    {!v.body}
</aura:component>
```

The client-side controller calls `$A.createComponent()` to create a `lightning:button` with a local ID (`aura:id`) and a handler for the `onclick` attribute. The `function(newButton, ...)` callback appends the button to the `body` of `c:createComponent`. The `newButton` that's dynamically created by `$A.createComponent()` is passed as the first argument to the callback.

```
/*createComponentController.js*/
({
    doInit : function(cmp) {
        $A.createComponent(
            "lightning:button",
            {
                "aura:id": "findableAuraId",
                "label": "Press Me",
                "onclick": cmp.getReference("c.handlePress")
            },
            function(newButton, status, errorMessage){
                //Add the new button to the body array
                if (status === "SUCCESS") {
                    var body = cmp.get("v.body");
                    body.push(newButton);
                    cmp.set("v.body", body);
                }
                else if (status === "INCOMPLETE") {
                    console.log("No response from server or client is offline.")
                    // Show offline error
                }
                else if (status === "ERROR") {
                    console.log("Error: " + errorMessage);
                    // Show error message
                }
            }
        );
    },

    handlePress : function(cmp) {
        // Find the button by the aura:id value
        console.log("button: " + cmp.find("findableAuraId"));
        console.log("button pressed");
    }
})
```

Note: `c:createComponent` contains a `{!v.body}` expression. When you use `cmp.set("v.body", ...)` to set the component body, you must explicitly include `{!v.body}` in your component markup.

Creating Nested Components

To dynamically create a component in the body of another component, use `$A.createComponents()` to create the components. In the function callback, nest the components by setting the inner component in the `body` of the outer component. This example creates a `lightning:icon` component in the `body` of a `lightning:card` component.

```
$A.createComponents([
    ["lightning:card",{
                "title" : "Dynamic Components"
            }],
            ["lightning:icon",{
                "iconName" : "utility:success",
                "alternativeText": "Icon that represents a successful step",
                "variant": "success",
                "class": "slds-m-around_small"
            }]
    ],
    function(components, status, errorMessages){
        if (status === "SUCCESS") {
            var card = components[0];
            var icon = components[1];
            // set lightning:icon as the body of lightning:card
            card.set("v.body", icon);
            cmp.set("v.body", card);
        }
        else if (status === "INCOMPLETE") {
            console.log("No response from server or client is offline.")
            // Show offline error
        }
        else if (status === "ERROR") {
            console.log("Error message: " + errorMessages[0].message);
        }
    }
);
```

Destroying Dynamically Created Components

After a component that is declared in markup is no longer in use, the framework automatically destroys it and frees up its memory.

If you create a component dynamically in JavaScript and don't add it to a facet like `v.body` or another attribute of type `Aura.Component[]`, you have to destroy it manually. Use `Component.destroy()` to destroy the component and free up its memory to avoid memory leaks.

Important: When a user navigates to a different page, components on the previous page remain in the cache and are hidden, not destroyed. See Event Handler Behavior for Active Components on page 283.

SEE ALSO:

- aura:dependency
- Invoking Actions on Component Initialization
- Dynamically Adding Event Handlers To a Component

Detecting Data Changes with Change Handlers

Configure a component to automatically invoke a change handler, which is a client-side controller action, when a value in one of the component's attributes changes.

When the value changes, the `valueChange.evt` event is automatically fired. The event has `type="VALUE"`.

In the component, define a handler with `name="change"`.

```
<aura:handler name="change" value="{!v.numItems}" action="{!c.itemsChange}"/>
```

The `value` attribute sets the component attribute that the change handler tracks.

The `action` attribute sets the client-side controller action to invoke when the attribute value changes.

A component can have multiple `<aura:handler name="change">` tags to detect changes to different attributes.

In the controller, define the action for the handler.

```
({
    itemsChange: function(cmp, evt) {
        console.log("numItems has changed");
        console.log("old value: " + evt.getParam("oldValue"));
        console.log("current value: " + evt.getParam("value"));
    }
})
```

The `valueChange` event gives you access to the previous value (`oldValue`) and the current value (`value`) in the handler action.

When a change occurs to a value that is represented by the `change` handler, the framework handles the firing of the event and rerendering of the component.

SEE ALSO:

Invoking Actions on Component Initialization

Finding Components by ID

Retrieve a component by its ID in JavaScript code.

Use `aura:id` to add a local ID of `button1` to the `lightning:button` component.

```
<lightning:button aura:id="button1" label="button1"/>
```

You can find the component by calling `cmp.find("button1")`, where `cmp` is a reference to the component containing the button. The `find()` function has one parameter, which is the local ID of a component within the markup.

`find()` returns different types depending on the result.

- If the local ID is unique, `find()` returns the component.
- If there are multiple components with the same local ID, `find()` returns an array of the components.
- If there is no matching local ID, `find()` returns `undefined`.

SEE ALSO:

Component IDs

Value Providers

Working with Attribute Values in JavaScript

These common patterns are useful for working with attribute values in JavaScript.

`component.get(String key)` and `component.set(String key, Object value)` retrieves and assigns values associated with the specified key on the component. Keys are passed in as an expression, which represents an attribute value.

To retrieve an attribute value of a component reference, use `component.find("cmpId").get("v.value")`.

Similarly, to set the attribute value of a component reference, use `component.find("cmpId").set("v.value", myValue)`.

This example shows how you can retrieve and set attribute values on a component reference, represented by the button with an ID of `button1`.

```
<aura:component>
    <aura:attribute name="buttonLabel" type="String"/>
    <lightning:button aura:id="button1" label="Button 1"/>
    {!v.buttonLabel}
    <lightning:button label="Get Label" onclick="{!c.getLabel}"/>
</aura:component>
```

This controller action retrieves the `label` attribute value of a button in a component and sets its value on the `buttonLabel` attribute.

```
({
    getLabel : function(component, event, helper) {
        var myLabel = component.find("button1").get("v.label");
        component.set("v.buttonLabel", myLabel);
    }
})
```

In the following examples, `cmp` is a reference to a component in your JavaScript code.

Get an Attribute Value

To get the value of a component's `label` attribute:

```
var label = cmp.get("v.label");
```

Set an Attribute Value

To set the value of a component's `label` attribute:

```
cmp.set("v.label","This is a label");
```

Deep Set an Attribute Value

If an attribute has an object or collection type, such as `Map`, you can deep set properties in the attribute value using the dot notation for expressions. For example, this code sets a value for the `firstName` property in the `user` attribute.

```
component.set("v.user.firstName", "Nina");
```

For deeply nested objects and attributes, continue adding dots to traverse the structure and access the nested values.

Let's look at a component with a `user` attribute of type `Map`.

```
<aura:component >
        <aura:attribute name="user" type="Map"
          default="{
            'id': 99,
            'firstName': 'Eunice',
            'lastName': 'Gomez'}" />

    <p>First Name: {!v.user.firstName}</p>

    <lightning:button onclick="{!c.deepSet}" label="Deep Set" />
</aura:component>
```

When you click the button in the component, you call the `deepSet` action in the client-side controller.

```
({
    deepSet : function(component, event, helper) {
        console.log(component.get("v.user.firstName"));
        component.set("v.user.firstName", "Nina");
        console.log(component.get("v.user.firstName"));
    }
})
```

The `component.set("v.user.firstName", "Nina")` line sets a value for the `firstName` property in the `user` attribute.

Validate That an Attribute Value Is Defined

To determine if a component's `label` attribute is defined:

```
var isDefined = !$A.util.isUndefined(cmp.get("v.label"));
```

Validate That an Attribute Value Is Empty

To determine if a component's `label` attribute is empty:

```
var isEmpty = $A.util.isEmpty(cmp.get("v.label"));
```

SEE ALSO:

Working with a Component Body in JavaScript

Working with a Component Body in JavaScript

These are useful and common patterns for working with a component's body in JavaScript.

In these examples, `cmp` is a reference to a component in your JavaScript code. It's usually easy to get a reference to a component in JavaScript code. Remember that the `body` attribute is an array of components, so you can use the JavaScript `Array` methods on it.

Note: When you use `cmp.set("v.body", ...)` to set the component body, you must explicitly include `{!v.body}` in your component markup.

Replace a Component's Body

To replace the current value of a component's body with another component:

```
// newCmp is a reference to another component
cmp.set("v.body", newCmp);
```

Clear a Component's Body

To clear or empty the current value of a component's body:

```
cmp.set("v.body", []);
```

Append a Component to a Component's Body

To append a `newCmp` component to a component's body:

```
var body = cmp.get("v.body");
// newCmp is a reference to another component
body.push(newCmp);
cmp.set("v.body", body);
```

Prepend a Component to a Component's Body

To prepend a `newCmp` component to a component's body:

```
var body = cmp.get("v.body");
body.unshift(newCmp);
cmp.set("v.body", body);
```

Remove a Component from a Component's Body

To remove an indexed entry from a component's body:

```
var body = cmp.get("v.body");
// Index (3) is zero-based so remove the fourth component in the body
body.splice(3, 1);
cmp.set("v.body", body);
```

SEE ALSO:

Component Body

Working with Attribute Values in JavaScript

Working with Events in JavaScript

These are useful and common patterns for working with events in JavaScript.

Events communicate data across components. Events can contain attributes with values set before the event is fired and read when the event is handled.

Fire an Event

Fire a component event or an application event that's registered on a component.

```
//Fire a component event
var compEvent = cmp.getEvent("sampleComponentEvent");
compEvent.fire();

//Fire an application event
var appEvent = $A.get("e.c:appEvent");
appEvent.fire();
```

For more information, see:

- Fire Component Events
- Fire Application Events

Get an Event Name

To get the name of the event that's fired:

```
event.getSource().getName();
```

Get an Event Parameter

To get an attribute that's passed into an event:

```
event.getParam("value");
```

Get Parameters on an Event

To get all attributes that are passed into an event:

```
event.getParams();
```

`event.getParams()` returns an object containing all event parameters.

Get the Current Phase of an Event

To get the current phase of an event:

```
event.getPhase();
```

If the event hasn't been fired, `event.getPhase()` returns `undefined`. Possible return values for component and application events are `capture`, `bubble`, and `default`. Value events return `default`. For more information, see:

- Component Event Propagation
- Application Event Propagation

Get the Source Component

To get the component that fired the event:

```
event.getSource();
```

To retrieve an attribute on the component that fired the event:

```
event.getSource().get("v.myName");
```

Pause the Event

To pause the fired event:

```
event.pause();
```

If paused, the event is not handled until `event.resume()` is called. You can pause an event in the `capture` or `bubble` phase only. For more information, see:

- Handling Bubbled or Captured Component Events
- Handling Bubbled or Captured Application Events

Prevent the Default Event Execution

To cancel the default action on the event:

```
event.preventDefault();
```

For example, you can prevent a `lightning:button` component from submitting a form when it's clicked.

Resume a Paused Event

To resume event handling for a paused event:

```
event.resume();
```

You can resume a paused event in the `capture` or `bubble` phase only. For more information, see:

- Handling Bubbled or Captured Component Events
- Handling Bubbled or Captured Application Events

Set a Value for an Event Parameter

To set a value for an event parameter:

```
event.setParam("name", cmp.get("v.myName"));
```

If the event has already been fired, setting a parameter value has no effect on the event.

Set Values for Event Parameters

To set values for parameters on an event:

```
event.setParams({
    key : value
});
```

If the event has already been fired, setting the parameter values has no effect on the event.

Stop Event Propagation

To prevent further propagation of an event:

```
event.stopPropagation();
```

You can stop event propagation in the `capture` or `bubble` phase only.

Modifying the DOM

The Document Object Model (DOM) is the language-independent model for representing and interacting with objects in HTML and XML documents. It's important to know how to modify the DOM safely so that the framework's rendering service doesn't stomp on your changes and give you unexpected results.

IN THIS SECTION:

Modifying DOM Elements Managed by the Aura Components Programming Model

The framework creates and manages the DOM elements owned by a component. If you want to modify these DOM elements created by the framework, modify the DOM elements in the handler for the component's `render` event or in a custom renderer. Otherwise, the framework will override your changes when the component is rerendered.

Modifying DOM Elements Managed by External Libraries

You can use different libraries, such as a charting library, to create and manage DOM elements. You don't have to modify these DOM elements within the `render` event handler or a renderer because they are managed by the external library.

Modifying DOM Elements Managed by the Aura Components Programming Model

The framework creates and manages the DOM elements owned by a component. If you want to modify these DOM elements created by the framework, modify the DOM elements in the handler for the component's `render` event or in a custom renderer. Otherwise, the framework will override your changes when the component is rerendered.

For example, if you modify DOM elements directly from a client-side controller, the changes may be overwritten when the component is rendered.

You can read from the DOM outside a `render` event handler or a custom renderer.

The simplest approach is to leave DOM updates to the framework. Update a component's attribute and use an expression in the markup. The framework's rendering service takes care of the DOM updates.

You can modify CSS classes for a component outside a renderer by using the `$A.util.addClass()`, `$A.util.removeClass()`, and `$A.util.toggleClass()` methods.

There are some use cases where you want to perform post-processing on the DOM or react to rendering or rerendering of a component. For these use cases, there are a few options.

IN THIS SECTION:

Handle the render Event

When a component is rendered or rerendered, the `aura:valueRender` event, also known as the `render` event, is fired. Handle this event to perform post-processing on the DOM or react to component rendering or rerendering. The event is preferred and easier to use than the alternative of creating a custom renderer.

Create a Custom Renderer

The framework's rendering service takes in-memory component state and creates and manages the DOM elements owned by the component. If you want to modify DOM elements created by the framework for a component, you can modify the DOM elements in the component's renderer. Otherwise, the framework will override your changes when the component is rerendered.

SEE ALSO:

Modifying DOM Elements Managed by External Libraries

Using Expressions

Dynamically Showing or Hiding Markup

Handle the `render` Event

When a component is rendered or rerendered, the `aura:valueRender` event, also known as the `render` event, is fired. Handle this event to perform post-processing on the DOM or react to component rendering or rerendering. The event is preferred and easier to use than the alternative of creating a custom renderer.

The `render` event is fired after all methods in a custom renderer are invoked. For more details on the sequence in the rendering or rerendering lifecycles, see Create a Custom Renderer.

Handling the `aura:valueRender` event is similar to handling the `init` hook. Add a handler to your component's markup.

```
<aura:handler name="render" value="{!this}" action="{!c.onRender}"/>
```

In this example, the `onRender` action in your client-side controller handles initial rendering and rerendering of the component. You can choose any name for the `action` attribute.

SEE ALSO:

Invoking Actions on Component Initialization

Create a Custom Renderer

Create a Custom Renderer

The framework's rendering service takes in-memory component state and creates and manages the DOM elements owned by the component. If you want to modify DOM elements created by the framework for a component, you can modify the DOM elements in the component's renderer. Otherwise, the framework will override your changes when the component is rerendered.

The DOM is the language-independent model for representing and interacting with objects in HTML and XML documents. The framework automatically renders your components so you don't have to know anything more about rendering unless you need to customize the default rendering behavior for a component.

Note: It's preferred and easier to handle the `render` event rather than the alternative of creating a custom renderer.

Base Component Rendering

The base component in the framework is `aura:component`. Every component extends this base component.

The renderer for `aura:component` is in `componentRenderer.js`. This renderer has base implementations for the four phases of the rendering and rerendering cycles:

- `render()`
- `rerender()`
- `afterRender()`
- `unrender()`

The framework calls these functions as part of the rendering and rerendering lifecycles and we will learn more about them soon. You can override the base rendering functions in a custom renderer.

Rendering Lifecycle

The rendering lifecycle happens once in the lifetime of a component unless the component gets explicitly unrendered. When you create a component:

1. The framework fires an `init` event, enabling you to update a component or fire an event after component construction but before rendering.
2. The `render()` method is called to render the component's body.
3. The `afterRender()` method is called to enable you to interact with the DOM tree after the framework's rendering service has inserted DOM elements.
4. The framework fires a `render` event, enabling you to interact with the DOM tree after the framework's rendering service has inserted DOM elements. Handling the `render` event is preferred to creating a custom renderer and overriding `afterRender()`.

Rerendering Lifecycle

The rerendering lifecycle automatically handles rerendering of components whenever the underlying data changes. Here is a typical sequence.

1. A browser event triggers one or more Lightning events.
2. Each Lightning event triggers one or more actions that can update data. The updated data can fire more events.
3. The rendering service tracks the stack of events that are fired.
4. The framework rerenders all the components that own modified data by calling each component's `rerender()` method.
5. The framework fires a `render` event, enabling you to interact with the DOM tree after the framework rerenders a component. Handling the `render` event is preferred to creating a custom renderer and overriding `rerender()`.

The component rerendering lifecycle repeats whenever the underlying data changes as long as the component is valid and not explicitly unrendered.

For more information, see Events Fired During the Rendering Lifecycle .

Custom Renderer

You don't normally have to write a custom renderer, but it's useful when you want to interact with the DOM tree after the framework's rendering service has inserted DOM elements. If you want to customize rendering behavior and you can't do it in markup or by using the `init` event, you can create a client-side renderer.

A renderer file is part of the component bundle and is auto-wired if you follow the naming convention, `<componentName>Renderer.js`. For example, the renderer for `sample.cmp` would be in `sampleRenderer.js`.

Note: These guidelines are important when you customize rendering.

- Only modify DOM elements that are part of the component. Never break component encapsulation by reaching in to another component and changing its DOM elements, even if you are reaching in from the parent component.
- Never fire an event as it can trigger new rendering cycles. An alternative is to use an `init` event instead.
- Don't set attribute values on other components as these changes can trigger new rendering cycles.
- Move as much of the UI concerns, including positioning, to CSS.

Customize Component Rendering

Customize rendering by creating a `render()` function in your component's renderer to override the base `render()` function, which updates the DOM.

The `render()` function returns a DOM node, an array of DOM nodes, or nothing. The base HTML component expects DOM nodes when it renders a component.

You generally want to extend default rendering by calling `superRender()` from your `render()` function before you add your custom rendering code. Calling `superRender()` creates the DOM nodes specified in the markup.

This code outlines a custom `render()` function.

```
render : function(cmp, helper) {
    var ret = this.superRender();
    // do custom rendering here
    return ret;
},
```

Rerender Components

When an event is fired, it may trigger actions to change data and call `rerender()` on affected components. The `rerender()` function enables components to update themselves based on updates to other components since they were last rendered. This function doesn't return a value.

If you update data in a component, the framework automatically calls `rerender()`.

You generally want to extend default rerendering by calling `superRerender()` from your `renderer()` function before you add your custom rerendering code. Calling `superRerender()` chains the rerendering to the components in the `body` attribute.

This code outlines a custom `rerender()` function.

```
rerender : function(cmp, helper){
    this.superRerender();
    // do custom rerendering here
}
```

Access the DOM After Rendering

The `afterRender()` function enables you to interact with the DOM tree after the framework's rendering service has inserted DOM elements. It's not necessarily the final call in the rendering lifecycle; it's simply called after `render()` and it doesn't return a value.

You generally want to extend default after rendering by calling `superAfterRender()` function before you add your custom code.

This code outlines a custom `afterRender()` function.

```
afterRender: function (component, helper) {
    this.superAfterRender();
    // interact with the DOM here
},
```

Unrender Components

The base `unrender()` function deletes all the DOM nodes rendered by a component's `render()` function. It is called by the framework when a component is being destroyed. Customize this behavior by overriding `unrender()` in your component's renderer. This method can be useful when you are working with third-party libraries that are not native to the framework.

You generally want to extend default unrendering by calling `superUnrender()` from your `unrender()` function before you add your custom code.

This code outlines a custom `unrender()` function.

```
unrender: function () {
    this.superUnrender();
    // do custom unrendering here
}
```

SEE ALSO:

Modifying the DOM

Invoking Actions on Component Initialization

Component Bundles

Modifying Components Outside the Framework Lifecycle

Sharing JavaScript Code in a Component Bundle

Modifying DOM Elements Managed by External Libraries

You can use different libraries, such as a charting library, to create and manage DOM elements. You don't have to modify these DOM elements within the `render` event handler or a renderer because they are managed by the external library.

A `render` event handler or a renderer are used only to customize DOM elements created and managed by the Aura Components programming model.

To use external libraries, use `<ltng:require>`. The `afterScriptsLoaded` attribute enables you to interact with the DOM after your libraries have loaded and the DOM is ready. `<ltng:require>` tag orchestrates the loading of your library of choice with the rendering cycle of the Aura Components programming model to ensure that everything works in concert.

SEE ALSO:

Using External JavaScript Libraries

Modifying DOM Elements Managed by the Aura Components Programming Model

Checking Component Validity

If you navigate elsewhere in the UI while asynchronous code is executing, the framework unrenders and destroys the component that made the asynchronous request. You can still have a reference to that component, but it is no longer valid. The `cmp.isValid()` call returns `false` for an invalid component.

If you call `cmp.get()` on an invalid component, `cmp.get()` returns `null`.

If you call `cmp.set()` on an invalid component, nothing happens and no error occurs. It's essentially a no op.

In many scenarios, the `cmp.isValid()` call isn't necessary because a `null` check on a value retrieved from `cmp.get()` is sufficient. The main reason to call `cmp.isValid()` is if you're making multiple calls against the component and you want to avoid a `null` check for each result.

Inside the Framework Lifecycle

You don't need a `cmp.isValid()` check in the callback in a client-side controller when you reference the component associated with the client-side controller. The framework automatically checks that the component is valid. Similarly, you don't need a `cmp.isValid()` check during event handling or in a framework lifecycle hook, such as the `init` event.

Let's look at a sample client-side controller.

```
({
    "doSomething" : function(cmp) {
        var action = cmp.get("c.serverEcho");
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                if (cmp.get("v.displayResult)) {
                    alert("From server: " + response.getReturnValue());
                }
            }
            // other state handling omitted for brevity
        });

        $A.enqueueAction(action);
    }
})
```

The component wired to the client-side controller is passed into the `doSomething` action as the `cmp` parameter. When `cmp.get("v.displayResult)` is called, we don't need a `cmp.isValid()` check.

However, if you hold a reference to another component that may not be valid despite your component being valid, you might need a `cmp.isValid()` check for the other component. Let's look at another example of a component that has a reference to another component with a local ID of `child`.

```
({
    "doSomething" : function(cmp) {
        var action = cmp.get("c.serverEcho");
        var child = cmp.find("child");
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                if (child.get("v.displayResult)) {
                    alert("From server: " + response.getReturnValue());
                }
            }
            // other state handling omitted for brevity
        });

        $A.enqueueAction(action);
    }
})
```

This line in the previous example without the child component:

```
if (cmp.get("v.displayResult)) {
```

changed to:

```
if (child.get("v.displayResult)) {
```

You don't need a `child.isValid()` call here as `child.get("v.displayResult)` will return `null` if the child component is invalid. Add a `child.isValid()` check only if you're making multiple calls against the child component and you want to avoid a `null` check for each result.

Outside the Framework Lifecycle

If you reference a component in asynchronous code, such as `setTimeout()` or `setInterval()`, or when you use Promises, a `cmp.isValid()` call checks that the component is still valid before processing the results of the asynchronous request. In many scenarios, the `cmp.isValid()` call isn't necessary because a `null` check on a value retrieved from `cmp.get()` is sufficient. The main reason to call `cmp.isValid()` is if you're making multiple calls against the component and you want to avoid a `null` check for each result.

For example, you don't need a `cmp.isValid()` check within this `setTimeout()` call as the `cmp.set()` call doesn't do anything when the component is invalid.

```
window.setTimeout(
    $A.getCallback(function() {
        cmp.set("v.visible", true);
    }), 5000
);
```

SEE ALSO:

- Handling Events with Client-Side Controllers
- Invoking Actions on Component Initialization
- Modifying Components Outside the Framework Lifecycle

Modifying Components Outside the Framework Lifecycle

Use `$A.getCallback()` to wrap any code that modifies a component outside the normal rerendering lifecycle, such as in a `setTimeout()` call. The `$A.getCallback()` call ensures that the framework rerenders the modified component and processes any enqueued actions.

Note: `$A.run()` is deprecated. Use `$A.getCallback()` instead.

You don't need to use `$A.getCallback()` if your code is executed as part of the framework's call stack; for example, your code is handling an event or in the callback for a server-side controller action. An exception is when you want to pass the callback to Lightning Data Service, such as when you are creating a record using `force:recordData`. If the callback is passed in without being wrapped in `$A.getCallback()`, any attempt to access private attributes of your component results in access check failures.

An example of where you need to use `$A.getCallback()` is calling `window.setTimeout()` in an event handler to execute some logic after a time delay. This puts your code outside the framework's call stack.

This sample sets the `visible` attribute on a component to `true` after a five-second delay.

```
window.setTimeout(
    $A.getCallback(function() {
        cmp.set("v.visible", true);
    }), 5000
);
```

Note how the code updating a component attribute is wrapped in `$A.getCallback()`, which ensures that the framework rerenders the modified component.

Note: You don't need a `cmp.isValid()` check within this `setTimeout()` call as the `cmp.set()` call doesn't do anything when the component is invalid.

Warning: Don't save a reference to a function wrapped in `$A.getCallback()`. If you use the reference later to send actions, the saved transaction state will cause the actions to be aborted.

SEE ALSO:

- Creating a Record
- Handling Events with Client-Side Controllers
- Checking Component Validity
- Firing Events from Non-Aura Code
- Communicating with Events

Throwing and Handling Errors

The framework gives you flexibility in handling unrecoverable and recoverable app errors in JavaScript code. For example, you can throw these errors in a callback when handling an error in a server-side response.

Note: Don't depend on the internals of a Lightning base component for error handling as its internals can change in future releases. Errors that are recoverable can change into unrecoverable errors and vice versa. If you encounter an unexpected error, you can sometimes get more information by enabling debug mode.

Unrecoverable Errors

Use `throw new Error("error message here")` for unrecoverable errors, such as an error that prevents your app from starting successfully. The error message is displayed.

Note: `$A.error()` is deprecated. Throw the native JavaScript `Error` object instead by using `throw new Error()`.

This example shows you the basics of throwing an unrecoverable error in a JavaScript controller.

```
<!--c:unrecoverableError-->
<aura:component>
    <lightning:button label="throw error" onclick="{!c.throwError}"/>
</aura:component>
```

Here is the client-side controller source.

```
/*unrecoverableErrorController.js*/
({
    throwError : function(component, event){
        throw new Error("I can’t go on. This is the end.");
    }
})
```

Recoverable Errors

To handle recoverable errors, use a component, such as `ui:message`, to tell users about the problem.

This sample shows you the basics of throwing and catching a recoverable error in a JavaScript controller.

```
<!--c:recoverableError-->
<aura:component>
    <p>Click the button to trigger the controller to throw an error.</p>
    <div aura:id="div1"></div>

    <lightning:button label="Throw an Error" onclick="{!c.throwErrorForKicks}"/>
</aura:component>
```

Here is the client-side controller source.

```
/*recoverableErrorController.js*/
({
    throwErrorForKicks: function(cmp) {
        // this sample always throws an error to demo try/catch
        var hasPerm = false;
        try {
            if (!hasPerm) {
                throw new Error("You don't have permission to edit this record.");
            }
        }
        catch (e) {
            $A.createComponents([
                ["ui:message",{
                    "title" : "Sample Thrown Error",
                    "severity" : "error",
                }],
                ["lightning:formattedText",{
                    "value" : e.message
                }]
                ],
                function(components, status, errorMessage){
                    if (status === "SUCCESS") {
                        var message = components[0];
                        var outputText = components[1];
                        // set the body of the ui:message to be the ui:outputText
                        message.set("v.body", outputText);
                        var div1 = cmp.find("div1");
                        // Replace div body with the dynamic component
                        div1.set("v.body", message);
                    }
                    else if (status === "INCOMPLETE") {
                        console.log("No response from server or client is offline.")
                        // Show offline error
                    }
                    else if (status === "ERROR") {
                        console.log("Error: " + errorMessage);
                        // Show error message
                    }
                }
            );
        }
    }
})
```

The controller code always throws an error and catches it in this example. The message in the error is displayed to the user in a dynamically created `ui:message` component. The body of the `ui:message` is a `ui:outputText` component containing the error text.

SEE ALSO:

Dynamically Creating Components

Calling Component Methods

Use `<aura:method>` to define a method as part of a component's API. This enables you to directly call a method in a component's client-side controller instead of firing and handling a component event. Using `<aura:method>` simplifies the code needed for a parent component to call a method on a child component that it contains.

Communicate Between Components

Use `aura:method` to communicate down the containment hierarchy. For example, a parent component calls an `aura:method` on a child component that it contains.

To communicate up the containment hierarchy, fire a component event in the child component and handle it in the parent component.

Syntax

Use this syntax to call a method in JavaScript code.

```
cmp.sampleMethod(arg1, … argN);
```

`cmp` is a reference to the component.

`sampleMethod` is the name of the `aura:method`.

`arg1, … argN` is an optional comma-separated list of arguments passed to the method. Each argument corresponds to an `aura:attribute` defined in the `aura:method` markup.

Using Inherited Methods

A sub component that extends a super component has access to any methods defined in the super component.

An interface can also include an `<aura:method>` tag. A component that implements the interface can access the method.

Example

Let's look at an example app.

```
<!-- c:auraMethodCallerWrapper.app -->
<aura:application >
    <c:auraMethodCaller />
</aura:application>
```

`c:auraMethodCallerWrapper.app` contains a `c:auraMethodCaller` component.

```
<!-- c:auraMethodCaller.cmp -->
<aura:component >
    <p>Parent component calls aura:method in child component</p>
    <c:auraMethod aura:id="child" />

    ...
</aura:component>
```

`c:auraMethodCaller` is the parent component. `c:auraMethodCaller` contains the child component, `c:auraMethod`.

We'll show how `c:auraMethodCaller` calls an `aura:method` defined in `c:auraMethod`.

We'll use `c:auraMethodCallerWrapper.app` to see how to return results from synchronous and asynchronous code.

IN THIS SECTION:

Return Result for Synchronous Code

`aura:method` executes synchronously. A synchronous method finishes executing before it returns. Use the `return` statement to return a value from synchronous JavaScript code.

Return Result for Asynchronous Code

aura:method executes synchronously. Use the return statement to return a value from synchronous JavaScript code. JavaScript code that calls a server-side action is asynchronous. Asynchronous code can continue to execute after it returns. You can't use the return statement to return the result of an asynchronous call because the aura:method returns before the asynchronous code completes. For asynchronous code, use a callback instead of a return statement.

SEE ALSO:

aura:method

Component Events

Return Result for Synchronous Code

aura:method executes synchronously. A synchronous method finishes executing before it returns. Use the return statement to return a value from synchronous JavaScript code.

An asynchronous method can continue to execute after it returns. JavaScript code often uses the callback pattern to return a result after asynchronous code completes. We'll describe later how to return a result for an asynchronous action.

Step 1: Define **aura:method** in Markup

Let's look at a logParam aura:method that executes synchronous code. We'll use the c:auraMethodCallerWrapper.app and components outlined in Calling Component Methods. Here's the markup that defines the aura:method.

```
<!-- c:auraMethod -->
<aura:component>
    <aura:method name="logParam"
      description="Sample method with parameter">
        <aura:attribute name="message" type="String" default="default message" />
    </aura:method>

    <p>This component has an aura:method definition.</p>
</aura:component>
```

The logParam aura:method has an aura:attribute with a name of message. This attribute enables you to set a message parameter when you call the logParam method.

The name attribute of logParam configures the aura:method to invoke logParam() in the client-side controller.

An aura:method can have multiple aura:attribute tags. Each aura:attribute corresponds to a parameter that you can pass into the aura:method. For more details on the syntax, see aura:method.

You don't explicitly declare a return value in the aura:method markup. You just use a return statement in the JavaScript controller.

Step 2: Implement **aura:method** Logic in Controller

The logParam aura:method invokes logParam() in auraMethodController.js. Let's look at that source.

```
/* auraMethodController.js */
({
    logParam : function(cmp, event) {
        var params = event.getParam('arguments');
        if (params) {
            var message = params.message;
            console.log("message: " + message);
            return message;
        }
    },
})
```

`logParam()` simply logs the parameter passed in and returns the parameter value to demonstrate how to use the `return` statement. If your code is synchronous, you can use a `return` statement; for example, you're not making an asynchronous server-side action call.

Step 3: Call `aura:method` from Parent Controller

`callAuraMethod()` in the controller for `c:auraMethodCaller` calls the `logParam` `aura:method` defined in its child component, `c:auraMethod`. Here's the controller for `c:auraMethodCaller`.

```
/* auraMethodCallerController.js */
({
    callAuraMethod : function(component, event, helper) {
        var childCmp = component.find("child");
        // call the aura:method in the child component
        var auraMethodResult =
          childCmp.logParam("message sent by parent component");
        console.log("auraMethodResult: " + auraMethodResult);
    },
})
```

`callAuraMethod()` finds the child component, `c:auraMethod`, and calls its `logParam` `aura:method` with an argument for the message parameter of the `aura:method`.

```
childCmp.logParam("message sent by parent component");
```

`auraMethodResult` is the value returned from `logParam`.

Step 4: Add Button to Initiate Call to `aura:method`

The `c:auraMethodCaller` markup contains a `lightning:button` that invokes `callAuraMethod()` in `auraMethodCallerController.js`. We use this button to initiate the call to `aura:method` in the child component.

```
<!-- c:auraMethodCaller.cmp -->
<aura:component >
    <p>Parent component calls aura:method in child component</p>
    <c:auraMethod aura:id="child" />

    <lightning:button label="Call aura:method in child component"
        onclick="{! c.callAuraMethod}" />
</aura:component>
```

SEE ALSO:

Return Result for Asynchronous Code

Calling Component Methods

aura:method

Return Result for Asynchronous Code

`aura:method` executes synchronously. Use the `return` statement to return a value from synchronous JavaScript code. JavaScript code that calls a server-side action is asynchronous. Asynchronous code can continue to execute after it returns. You can't use the `return` statement to return the result of an asynchronous call because the `aura:method` returns before the asynchronous code completes. For asynchronous code, use a callback instead of a `return` statement.

Step 1: Define **`aura:method`** in Markup

Let's look at an `echo` `aura:method` that uses a callback. We'll use the `c:auraMethodCallerWrapper.app` and components outlined in Calling Component Methods. Here's the `echo` `aura:method` in the `c:auraMethod` component.

```
<!-- c:auraMethod -->
<aura:component controller="SimpleServerSideController">
    <aura:method name="echo"
      description="Sample method with server-side call">
        <aura:attribute name="callback" type="Function" />
    </aura:method>

    <p>This component has an aura:method definition.</p>
</aura:component>
```

The `echo` `aura:method` has an `aura:attribute` with a name of callback. This attribute enables you to set a callback that's invoked by the `aura:method` after execution of the server-side action in `SimpleServerSideController`.

Step 2: Implement **`aura:method`** Logic in Controller

The `echo` `aura:method` invokes `echo()` in `auraMethodController.js`. Let's look at the source.

```
/* auraMethodController.js */
({
    echo : function(cmp, event) {
        var params = event.getParam('arguments');
        var callback;
        if (params) {
            callback = params.callback;
        }

        var action = cmp.get("c.serverEcho");
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                console.log("From server: " + response.getReturnValue());
                // return doesn't work for async server action call
                //return response.getReturnValue();
                // call the callback passed into aura:method
                if (callback) callback(response.getReturnValue());
            }
            else if (state === "INCOMPLETE") {
                // do something
            }
            else if (state === "ERROR") {
                var errors = response.getError();
                if (errors) {
                    if (errors[0] && errors[0].message) {
                        console.log("Error message: " +
                          errors[0].message);
                    }
                } else {
                    console.log("Unknown error");
                }
            }
        });
        $A.enqueueAction(action);
    },
})
```

`echo()` calls the `serverEcho()` server-side controller action, which we'll create next.

Note: You can't return the result with a `return` statement. The `aura:method` returns before the asynchronous server-side action call completes. Instead, we invoke the callback passed into the `aura:method` and set the result as a parameter in the callback.

Step 3: Create Apex Server-Side Controller

The `echo` `aura:method` calls a server-side controller action called `serverEcho`. Here's the source for the server-side controller.

```
public with sharing class SimpleServerSideController {
    @AuraEnabled
    public static String serverEcho() {
        return ('Hello from the server');
    }
}
```

The `serverEcho()` method returns a `String`.

Step 4: Call **`aura:method`** from Parent Controller

Here's the controller for `c:auraMethodCaller`. It calls the `echo` `aura:method` in its child component, `c:auraMethod`.

```
/* auraMethodCallerController.js */
({
    callAuraMethodServerTrip : function(component, event, helper) {
        var childCmp = component.find("child");
        // call the aura:method in the child component
        childCmp.echo(function(result) {
            console.log("callback for aura:method was executed");
            console.log("result: " + result);
        });
    },
})
```

`callAuraMethodServerTrip()` finds the child component, `c:auraMethod`, and calls its `echo` `aura:method`. `echo()` passes a callback function into the `aura:method`.

The callback configured in `auraMethodCallerController.js` logs the result.

```
function(result) {
    console.log("callback for aura:method was executed");
    console.log("result: " + result);
}
```

Step 5: Add Button to Initiate Call to `aura:method`

The `c:auraMethodCaller` markup contains a `lightning:button` that invokes `callAuraMethodServerTrip()` in `auraMethodCallerController.js`. We use this button to initiate the call to the `aura:method` in the child component.

Here's the markup for `c:auraMethodCaller`.

```
<!-- c:auraMethodCaller.cmp -->
<aura:component >
    <p>Parent component calls aura:method in child component</p>
    <c:auraMethod aura:id="child" />

    <lightning:button label="Call aura:method (server trip) in child component"
        onclick="{! c.callAuraMethodServerTrip}" />
</aura:component>
```

SEE ALSO:

Return Result for Synchronous Code

Calling Component Methods

aura:method

Dynamically Adding Event Handlers To a Component

You can dynamically add a handler for an event that a component fires.

The `addEventHandler()` method in the `Component` object replaces the deprecated `addHandler()` method.

To add an event handler to a component dynamically, use the `addEventHandler()` method.

```
addEventHandler(String event, Function handler, String phase, String includeFacets)
```

`event`

The first argument is the name of the event that triggers the handler. You can't force a component to start firing events that it doesn't fire, so make sure that this argument corresponds to an event that the component fires. The `<aura:registerEvent>` tag in a component's markup advertises an event that the component fires.

- For a component event, set this argument to match the `name` attribute of the `<aura:registerEvent>` tag.
- For an application event, set this argument to match the event descriptor in the format `namespace:eventName`.

`handler`

The second argument is the action that handles the event. The format is similar to the value you would put in the `action` attribute in the `<aura:handler>` tag if the handler was statically defined in the markup. There are two options for this argument.

- To use a controller action, use the format: `cmp.getReference("c.actionName")`.

- To use an anonymous function, use the format:

  ```
  function(auraEvent) {
      // handling logic here
  }
  ```

For a description of the other arguments, see the JavaScript API in the Aura Reference app.

You can also add an event handler to a component that is created dynamically in the callback function of `$A.createComponent()`. For more information, see Dynamically Creating Components.

Example

This component has buttons to fire and handle a component event and an application event.

```
<!--c:dynamicHandler-->
<aura:component >
    <aura:registerEvent name="compEvent" type="c:sampleEvent"/>
    <aura:registerEvent name="appEvent" type="c:appEvent"/>
    <h1>Add dynamic handler for event</h1>
    <p>
        <lightning:button label="Fire component event" onclick="{!c.fireEvent}" />
        <lightning:button label="Add dynamic event handler for component event"
onclick="{!c.addEventHandler}" />
    </p>
    <p>
        <lightning:button label="Fire application event" onclick="{!c.fireAppEvent}" />
        <lightning:button label="Add dynamic event handler for application event"
onclick="{!c.addAppEventHandler}" />
    </p>

</aura:component>
```

Here's the client-side controller.

```
/* dynamicHandlerController.js */
({
    fireEvent : function(cmp, event) {
        // Get the component event by using the
        // name value from <aura:registerEvent> tag
        var compEvent = cmp.getEvent("compEvent");
        compEvent.fire();
        console.log("Fired a component event");
    },

    addEventHandler : function(cmp, event) {
        // First param matches name attribute in <aura:registerEvent> tag
        cmp.addEventHandler("compEvent", cmp.getReference("c.handleEvent"));
        console.log("Added handler for component event");
    },

    handleEvent : function(cmp, event) {
        alert("Handled the component event");
    },
```

```
    fireAppEvent : function(cmp, event) {
        var appEvent = $A.get("e.c:appEvent");
        appEvent.fire();
        console.log("Fired an application event");
    },

    addAppEventHandler : function(cmp, event) {
        // Can use cmp.getReference() or anonymous function for handler
        // First param is event descriptor, "c:appEvent", for application events
        cmp.addEventHandler("c:appEvent", cmp.getReference("c.handleAppEvent"));
        // Can alternatively use anonymous function for handler
        //cmp.addEventHandler("c:appEvent", function(auraEvent) {
            // console.log("Handled the application event in anonymous function");
        //});
        console.log("Added handler for application event");
    },

    handleAppEvent : function(cmp, event) {
        alert("Handled the application event");
    }
})
```

Notice the first parameter of the `addEventHandler()` calls. The syntax for a component event is:

```
cmp.addEventHandler("compEvent", cmp.getReference("c.handleEvent"));
```

The syntax for an application event is:

```
cmp.addEventHandler("c:appEvent", cmp.getReference("c.handleAppEvent"));
```

For either a component or application event, you can use an anonymous function as a handler instead of using `cmp.getReference()` for a controller action.

For example, the application event handler could be:

```
cmp.addEventHandler("c:appEvent", function(auraEvent) {
    // add handler logic here
    console.log("Handled the application event in anonymous function");
});
```

SEE ALSO:

- Handling Events with Client-Side Controllers
- Handling Component Events
- Component Library

Dynamically Showing or Hiding Markup

You can use CSS to toggle markup visibility. However, `<aura:if>` is the preferred approach because it defers the creation and rendering of the enclosed element tree until needed.

For an example using `<aura:if>`, see Best Practices for Conditional Markup.

This example uses `$A.util.toggleClass(cmp, 'class')` to toggle visibility of markup.

```
<!--c:toggleCss-->
<aura:component>
    <lightning:button label="Toggle" onclick="{!c.toggle}"/>
    <p aura:id="text">Now you see me</p>
</aura:component>
```

```
/*toggleCssController.js*/
({
    toggle : function(component, event, helper) {
        var toggleText = component.find("text");
        $A.util.toggleClass(toggleText, "toggle");
    }
})
```

```
/*toggleCss.css*/
.THIS.toggle {
    display: none;
}
```

Note: There's no space in the `.THIS.toggle` selector because we're using the rule to match a `<p>` tag, which is a top-level element. For more information, see CSS in Components.

Add the `c:toggleCss` component to an app. To hide or show the text by toggling the CSS class, click the **Toggle** button.

SEE ALSO:

Handling Events with Client-Side Controllers

Component Attributes

Adding and Removing Styles

Adding and Removing Styles

You can add or remove a CSS style on a component or element during runtime.

To retrieve the class name on a component, use `component.find('myCmp').get('v.class')`, where `myCmp` is the `aura:id` attribute value.

To append and remove CSS classes from a component or element, use the `$A.util.addClass(cmpTarget, 'class')` and `$A.util.removeClass(cmpTarget, 'class')` methods.

Component source

```
<aura:component>
    <div aura:id="changeIt">Change Me!</div><br />
    <lightning:button onclick="{!c.applyCSS}" label="Add Style" />
    <lightning:button onclick="{!c.removeCSS}" label="Remove Style" />
</aura:component>
```

CSS source

```
.THIS.changeMe {
    background-color:yellow;
    width:200px;
}
```

Client-side controller source

```
{
    applyCSS: function(cmp, event) {
        var cmpTarget = cmp.find('changeIt');
        $A.util.addClass(cmpTarget, 'changeMe');
    },

    removeCSS: function(cmp, event) {
        var cmpTarget = cmp.find('changeIt');
        $A.util.removeClass(cmpTarget, 'changeMe');
    }
}
```

The buttons in this demo are wired to controller actions that append or remove the CSS styles. To append a CSS style to a component, use `$A.util.addClass(cmpTarget, 'class')`. Similarly, remove the class by using `$A.util.removeClass(cmpTarget, 'class')` in your controller. `cmp.find()` locates the component using the local ID, denoted by `aura:id="changeIt"` in this demo.

Toggling a Class

To toggle a class, use `$A.util.toggleClass(cmp, 'class')`, which adds or removes the class.

The `cmp` parameter can be component or a DOM element.

Note: We recommend using a component instead of a DOM element. If the utility function is not used inside `afterRender()` or `rerender()`, passing in `cmp.getElement()` might result in your class not being applied when the components are rerendered. For more information, see Events Fired During the Rendering Lifecycle on page 292.

To hide or show markup dynamically, see Dynamically Showing or Hiding Markup on page 370.

To conditionally set a class for an array of components, pass in the array to `$A.util.toggleClass()`.

```
mapClasses: function(arr, cssClass) {
    for(var cmp in arr) {
        $A.util.toggleClass(arr[cmp], cssClass);
    }
}
```

SEE ALSO:

- Handling Events with Client-Side Controllers
- CSS in Components
- Component Bundles

Which Button Was Pressed?

To find out which button was pressed in a component containing multiple buttons, use `Component.getLocalId()`.

Let's look at an example with multiple `lightning:button` components. Each button has a unique local ID, set by an `aura:id` attribute.

```
<!--c:buttonPressed-->
<aura:component>
    <aura:attribute name="whichButton" type="String" />

    <p>You clicked: {!v.whichButton}</p>

    <lightning:button aura:id="button1" label="Click me" onclick="{!c.nameThatButton}"/>
   <lightning:button aura:id="button2" label="Click me too" onclick="{!c.nameThatButton}"/>
</aura:component>
```

Use `event.getSource()` in the client-side controller to get the button component that was clicked. Call `getLocalId()` to get the `aura:id` of the clicked button.

```
/* buttonPressedController.js */
({
    nameThatButton : function(cmp, event, helper) {
        var whichOne = event.getSource().getLocalId();
        console.log(whichOne);
        cmp.set("v.whichButton", whichOne);
    }
})
```

In the client-side controller, you can use one of the following methods to find out which button was clicked.

- `event.getSource().getLocalId()` returns the `aura:id` of the clicked button.
- `event.getSource().get("v.name")` returns the `name` of the clicked button.

SEE ALSO:

Component IDs

Finding Components by ID

Formatting Dates in JavaScript

The `AuraLocalizationService` JavaScript API provides methods for formatting and localizing dates.

For example, the `formatDate()` method formats a date based on the `formatString` parameter set as the second argument.

```
formatDate (String | Number | Date date, String formatString)
```

The `date` parameter can be a String, Number, or most typically a JavaScript Date. If you provide a String value, use ISO 8601 format to avoid parsing warnings.

The `formatString` parameter contains tokens to format a date and time. For example, `"YYYY-MM-DD"` formats `15th January, 2017` as `"2017-01-15"`. The default format string comes from the `$Locale` value provider.

This table shows the list of tokens supported in `formatString`.

Description	Token	Output
Day of month	d	1 … 31
Day of month	dd	01 … 31

Description	**Token**	**Output**
Day of month. Deprecated. Use `dd`, which is identical.	DD	01 … 31
Day of week (number)	E	0 … 6
Day of week (short name)	EEE	Sun … Sat
Day of week (long name)	EEEE	Sunday … Saturday
Month	M	1 ... 12
Month	MM	01 … 12
Month (short name)	MMM	Jan … Dec
Month (full name)	MMMM	January … December
Year (two digits)	yy	17
Year (four digits)	yyyy	2017
Year. Deprecated. Use `yyyy`, which is identical.	y	2017
Year. Deprecated. Use `yyyy`, which is identical.	Y	2017
Year. Deprecated. Use `yy`, which is identical.	YY	17
Year. Deprecated. Use `yyyy`, which is identical.	YYYY	2017
Hour of day (1-12)	h	1 … 12
Hour of day (0-23)	H	0 … 23
Hour of day (00-23)	HH	00 … 23
Hour of day (1-24)	k	1 … 24
Hour of day (01-24)	kk	01 … 24
Minute	m	0 … 59
Minute	mm	00 … 59
Second	s	0 … 59
Second	ss	00 … 59
Fraction of second	SSS	000 … 999
AM or PM	a	AM or PM
AM or PM. Deprecated. Use `a`, which is identical.	A	AM or PM
Zone offset from UTC	Z	-12:00 … +14:00
Quarter of year	Q	1 … 4
Week of year	w	1 … 53

Description	Token	Output
Week of year	ww	01 … 53

There are similar methods that differ in their default output values.

- `formatDateTime()`—The default formatString outputs datetime instead of date.
- `formatDateTimeUTC()`—Formats a datetime in UTC standard time.
- `formatDateUTC()`—Formats a date in UTC standard time.

For more information on all the methods in `AuraLocalizationService`, see JavaScript API.

Example: This example converts a selected date on a date field using the given format, `yyyy-MM-dd`. The converted date is displayed below the date field.

```
<aura:component implements="flexipage:availableForRecordHome">
    <aura:attribute name="formatDate" type="String"/>
    <lightning:input
        type="date"
        value="{!v.formatDate}"
        onchange="{!c.convertDate}">
    </lightning:input>
    {!v.formatDate}
</aura:component>
```

```
({
    convertDate: function (cmp, event) {
        var date = event.getParam("value");
        var formatted = $A.localizationService.formatDate(date, "yyyy-MM-dd");
        cmp.set("v.formatDate", formatted);
    },
})
```

SEE ALSO:

Localization

Using JavaScript Promises

You can use ES6 Promises in JavaScript code. Promises can simplify code that handles the success or failure of asynchronous calls, or code that chains together multiple asynchronous calls.

If the browser doesn't provide a native version, the framework uses a polyfill so that promises work in all browsers supported for Lightning Experience.

We assume that you are familiar with the fundamentals of promises. For a great introduction to promises, see https://web.dev/articles/promises.

Promises are an optional feature. Some people love them, some don't. Use them if they make sense for your use case.

Create a Promise

This `firstPromise` function returns a Promise.

```
firstPromise : function() {
    return new Promise($A.getCallback(function(resolve, reject) {
      // do something

      if (/* success */) {
        resolve("Resolved");
      }
      else {
        reject("Rejected");
      }
    }));
}
```

The promise constructor determines the conditions for calling `resolve()` or `reject()` on the promise.

Chaining Promises

When you need to coordinate or chain together multiple callbacks, promises can be useful. The generic pattern is:

```
firstPromise()
    .then(
        // resolve handler
        $A.getCallback(function(result) {
            return anotherPromise();
        }),

        // reject handler
        $A.getCallback(function(error) {
            console.log("Promise was rejected: ", error);
            return errorRecoveryPromise();
        })
    )
    .then(
        // resolve handler
        $A.getCallback(function() {
            return yetAnotherPromise();
        })
    );
```

The `then()` method chains multiple promises. In this example, each resolve handler returns another promise.

`then()` is part of the Promises API. It takes two arguments:

1. A callback for a fulfilled promise (resolve handler)
2. A callback for a rejected promise (reject handler)

The first callback, `function(result)`, is called when `resolve()` is called in the promise constructor. The `result` object in the callback is the object passed as the argument to `resolve()`.

The second callback, `function(error)`, is called when `reject()` is called in the promise constructor. The `error` object in the callback is the object passed as the argument to `reject()`.

Note: The two callbacks are wrapped by `$A.getCallback()` in our example. What's that all about? Promises execute their resolve and reject functions asynchronously so the code is outside the Lightning event loop and normal rendering lifecycle. If the resolve or reject code makes any calls to the Lightning Component framework, such as setting a component attribute, use `$A.getCallback()` to wrap the code. For more information, see Modifying Components Outside the Framework Lifecycle on page 360.

Always Use `catch()` or a Reject Handler

The reject handler in the first `then()` method returns a promise with `errorRecoveryPromise()`. Reject handlers are often used "midstream" in a promise chain to trigger an error recovery mechanism.

The Promises API includes a `catch()` method to optionally catch unhandled errors. Always include a reject handler or a `catch()` method in your promise chain.

Throwing an error in a promise doesn't trigger `window.onerror`, which is where the framework configures its global error handler. If you don't have a `catch()` method, keep an eye on your browser's console during development for reports about uncaught errors in a promise. To show an error message in a `catch()` method, use `$A.reportError()`. The syntax for `catch()` is:

```
promise.then(...)
    .catch(function(error) {
        $A.reportError("error message here", error);
    });
```

For more information on `catch()`, see the Mozilla Developer Network.

Don't Use Storable Actions in Promises

The framework stores the response for storable actions in client-side cache. This stored response can dramatically improve the performance of your app and allow offline usage for devices that temporarily don't have a network connection. Storable actions are only suitable for read-only actions.

Storable actions might have their callbacks invoked more than once: first with cached data, then with updated data from the server. The multiple invocations don't align well with promises, which are expected to resolve or reject only once.

SEE ALSO:

Storable Actions

Making API Calls from Components

By default, you can't make calls to third-party APIs from client-side code. Add a remote site as a Trusted URL with Content Security Policy (CSP) directives to allow client-side component code to load assets from and make API requests to that site's domain.

The Lightning Component framework uses Content Security Policy (CSP) to impose restrictions on content. The main objective is to help prevent cross-site scripting (XSS) and other code injection attacks. Lightning apps are served from a different domain than Salesforce APIs, and the default CSP policy doesn't allow API calls from JavaScript code. You change the policy, and the content of the CSP header, by adding Trusted URLs.

Important: Otherwise, you can't load JavaScript resources from a third party, even if it's a trusted URL. To use a JavaScript library from a third-party site, add that third-party site to a static resource, and then add the static resource to your component. After the library is loaded from the static resource, you can use it as normal.

Sometimes, you have to make API calls from server-side controllers rather than client-side code. In particular, you can't make calls to Salesforce APIs from client-side Aura component code. For information about making API calls from server-side controllers, see Making API Calls from Apex on page 446.

SEE ALSO:

Security for Lightning Components: Content Security Policy Overview

Manage Trusted URLs

Control Access to Browser Features

To control whether requests to an external (non-Salesforce) server or URL can access the user's camera and microphone, enable the Permissions-Policy HTTP header. Then select when to allow access to each of these browser features.

1. From Setup, in the Quick Find box, enter `Session Settings`, and then select **Session Settings**.
2. In the Browser Feature Permissions section, select **Include Permissions-Policy HTTP header**. When this setting is disabled, all external apps and websites loaded from Salesforce can access the user's camera and microphone.
3. For Camera and Microphone, select when requests from Salesforce can access the browser feature.
   a. For the most granular control over access to this browser feature, select **Trusted URLs Only**.
      After you select this recommended setting, specify trusted URLs and the browser features that they can access from the Trusted URLs Setup page.
   b. To grant access to this browser feature for all external apps and websites loaded from Salesforce, select **Always**.
   c. To block access to the browser feature for all external apps and websites loaded from Salesforce, select **Never**.
      If you select Never, even scripts from Salesforce domains can't access the browser feature.
4. 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 modify session security settings:

- Customize Application

SEE ALSO:

Manage Trusted URLs

Manage Trusted URLs

Specify the URLs that you trust to interact with your users and network. Use Content Security Policy (CSP) directives to control the types of resources that Lightning components, third-party APIs, and WebSocket connections can load from each trusted URL. If you enabled the Permissions-Policy HTTP header in Session Settings, you can also control which URLs can access browser features from Salesforce.

For each trusted URL in Setup, you can specify CSP directives and Permissions-Policy directives. To specify the external URLs to which users can be redirected from Salesforce, see Manage Redirections to External URLs. To allow external sites to load your Visualforce pages or surveys in an inline frame (iframe), see Specify Trusted Domains for Inline Frames.

Note: To support integration across Salesforce products, Salesforce includes URLs in each CSP directive, even though those URLs aren't defined as trusted URLs. Salesforce regularly updates those URLs based on the latest requirements.

EDITIONS

Available in: Salesforce Classic and Lightning Experience

Available in: **Enterprise**, **Performance**, **Developer**, and **Unlimited** Editions

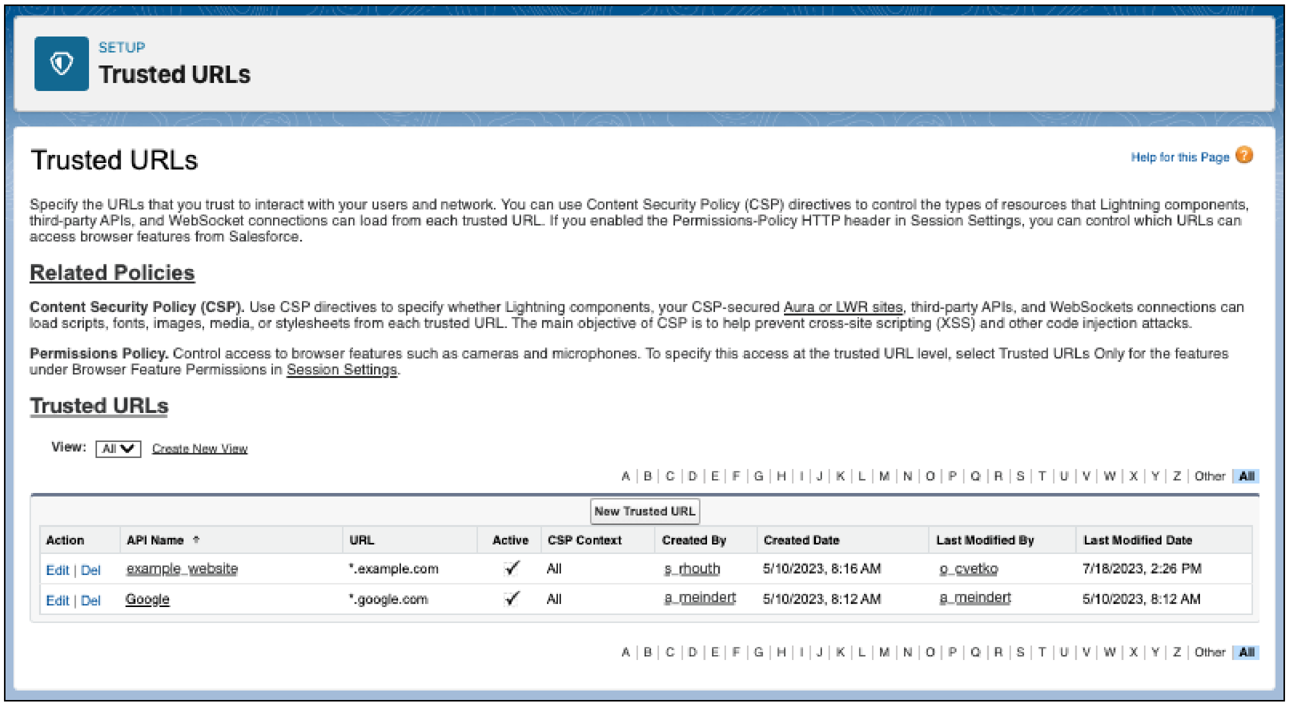
USER PERMISSIONS

To create, read, update, and delete trusted URLs:
- Customize Application AND Modify All Data

Add or Edit a Trusted URL

For each trusted URL in Setup, you can specify Content Security Policy (CSP) directives and Permissions-Policy directives.

1. From Setup, in the Quick Find box, enter `Trusted URLs`, and then select **Trusted URLs**.



Trusted URLs

Help for this Page

Specify the URLs that you trust to interact with your users and network. You can use Content Security Policy (CSP) directives to control the types of resources that Lightning components, third-party APIs, and WebSocket connections can load from each trusted URL. If you enabled the Permissions-Policy HTTP header in Session Settings, you can control which URLs can access browser features from Salesforce.

Related Policies

Content Security Policy (CSP). Use CSP directives to specify whether Lightning components, your CSP-secured Aura or LWR sites, third-party APIs, and WebSockets connections can load scripts, fonts, images, media, or stylesheets from each trusted URL. The main objective of CSP is to help prevent cross-site scripting (XSS) and other code injection attacks.

Permissions Policy. Control access to browser features such as cameras and microphones. To specify this access at the trusted URL level, select Trusted URLs Only for the features under Browser Feature Permissions in Session Settings.

Trusted URLs

View: All Create New View

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | Other | All

New Trusted URL

Action	API Name ↑	URL	Active	CSP Context	Created By	Created Date	Last Modified By	Last Modified Date
Edit | Del	example_website	*.example.com	✓	All	s_rhouth	5/10/2023, 8:16 AM	g_cvetko	7/18/2023, 2:26 PM
Edit | Del	Google	*.google.com	✓	All	a_meindert	5/10/2023, 8:12 AM	a_meindert	5/10/2023, 8:12 AM

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | Other | All

2. To add a new trusted URL, click **New Trusted URL**.
3. To edit an existing trusted URL, click **Edit**.
4. If you're adding a trusted URL, enter the API Name.

Enter only underscores and alphanumeric characters. The name must be unique, begin with a letter, not include spaces, not end with an underscore, and not contain two consecutive underscores.

If you edit the API name of an existing trusted URL, review your code and update references to the previous API name.

5. Edit or enter the URL.

   The trusted URL must include a domain name and can include a port. For example, https://example.com or https://example.com:8080.

   To reduce repetition, you can use the wildcard character * (asterisk). For example, *.example.com.

   For a third-party API, the URL must begin with https://. For example, https://example.com.

   For a WebSocket connection, the URL must begin with wss://. For example, wss://example.com.

   The host section of the URL can include an asterisk (`*`) as a wildcard. Otherwise, the URL cannot be malformed. Examples of malformed URLs that fail a syntax check are `malformed^url.example.com`, and `https://{subdomain}.example.com`.

6. Optionally, enter or edit a description for the trusted URL.
7. Optionally, to temporarily disable this trusted URL, deselect **Active**.
8. Specify at least one CSP directive or permissions policy directive for the trusted URL, and then save your changes.

Specify CSP Directives for a Trusted URL

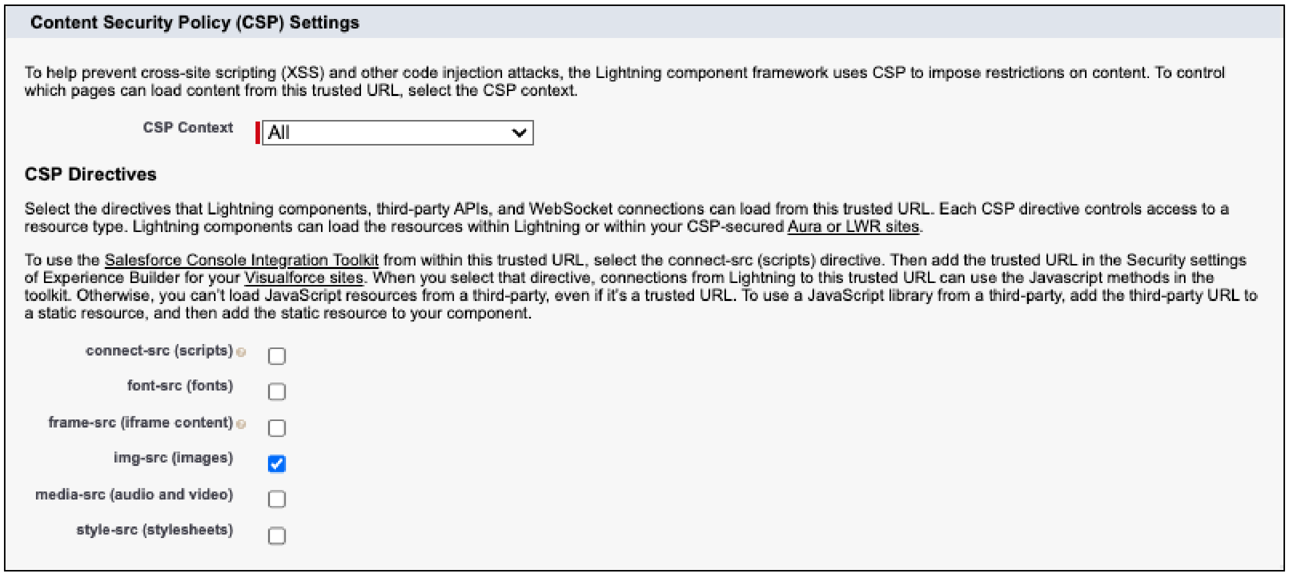
To help prevent cross-site scripting (XSS) and other code injection attacks, the Lightning component framework uses Content Security Policy (CSP) to impose restrictions on content. By default, the framework's headers allow content to be loaded only from secure (HTTPS) URLs and forbid XHR requests from JavaScript. To use third-party APIs that make requests to an external (non-Salesforce) server or to use a WebSocket connection, add the server as a Trusted URL.

To enable the corresponding access for Apex, create a remote site.

Note: Not every browser enforces CSP. For a list of browsers that enforce CSP, see `caniuse.com`

1. From Setup, in the Quick Find box, enter *`Trusted URLs`*, and then select **Trusted URLs**.

   You define the CSP context and directives in the Content Security Policy (CSP) Settings section of the Trusted URL page.



Content Security Policy (CSP) Settings

To help prevent cross-site scripting (XSS) and other code injection attacks, the Lightning component framework uses CSP to impose restrictions on content. To control which pages can load content from this trusted URL, select the CSP context.

CSP Context: All

CSP Directives

Select the directives that Lightning components, third-party APIs, and WebSocket connections can load from this trusted URL. Each CSP directive controls access to a resource type. Lightning components can load the resources within Lightning or within your CSP-secured Aura or LWR sites.

To use the Salesforce Console Integration Toolkit from within this trusted URL, select the connect-src (scripts) directive. Then add the trusted URL in the Security settings of Experience Builder for your Visualforce sites. When you select that directive, connections from Lightning to this trusted URL can use the Javascript methods in the toolkit. Otherwise, you can't load JavaScript resources from a third-party, even if it's a trusted URL. To use a JavaScript library from a third-party, add the third-party URL to a static resource, and then add the static resource to your component.

connect-src (scripts)	☐
font-src (fonts)	☐
frame-src (iframe content)	☐
img-src (images)	✓
media-src (audio and video)	☐
style-src (stylesheets)	☐

2. To control which pages can load content from this trusted URL, select the CSP context.
   a. To apply the CSP directives to all supported context types, select **All**. This context is the default.
   b. To apply the CSP directives to Experience Cloud sites only, select **Experience Builder Sites**.
   c. To apply the CSP directives to Lightning Experience pages only, select **Lightning Experience pages**.
   d. To apply the CSP directives to your custom Visualforce pages only, select **Visualforce Pages**.

   For custom Visualforce pages, content is restricted to CSP Trusted Sites only if the page's `cspHeader` attribute is set to `true`.

   Tip: To specify CSP directives for one URL with two of the three CSP contexts, create two trusted URL records with different API names.

3. Select the CSP directives for this trusted URL. Each CSP directive controls access to a resource type. Lightning components can load the resources within Lightning or within your CSP-secured Aura or LWR sites.
   a. To allow Lightning components, third-party APIs, and WebSocket connections to load URLs that use script interfaces from this trusted URL, select **connect-src (scripts)**.

      Note: To use the Salesforce Console Integration Toolkit from within a trusted URL, also add the trusted URL in the Security settings of Experience Builder for your Visualforce sites. Otherwise, you can't load JavaScript resources from a third party, even if it's a trusted URL.

      To use a JavaScript library from a third party, add the library to a static resource, and then add the static resource to your component.

   b. To allow Lightning components, third-party APIs, and WebSocket connections to load fonts from this trusted URL, select **font-src (fonts)**.
   c. To allow Lightning components, third-party APIs, and WebSocket connections to load resources contained in `<iframe>` elements from this trusted URL, select **frame-src (iframe content)**.
   d. To allow Lightning components, third-party APIs, and WebSocket connections to load images from this trusted URL, select **img-src (images)**. This option is enabled by default.
   e. To allow Lightning components, third-party APIs, and WebSocket connections to load audio and video from this trusted URL, select **media-src (audio and video)**.
   f. To allow Lightning components, third-party APIs, and WebSocket connections to load style sheets from this trusted URL, select **style-src (stylesheets)**.

4. After you save your changes, validate the header size for your Aura sites.
   For Aura sites in Experience Cloud, if the HTTP header size is greater than 8 KB, the directives are moved from the CSP header to the `<meta>` tag. To avoid errors from infrastructure limits, we recommend that the header size doesn't exceed 3 KB per CSP context.

SEE ALSO:

Configure Remote Site Settings

Secure Coding Guide: Secure Coding WebSockets

Security for Lightning Components: Content Security Policy Overview

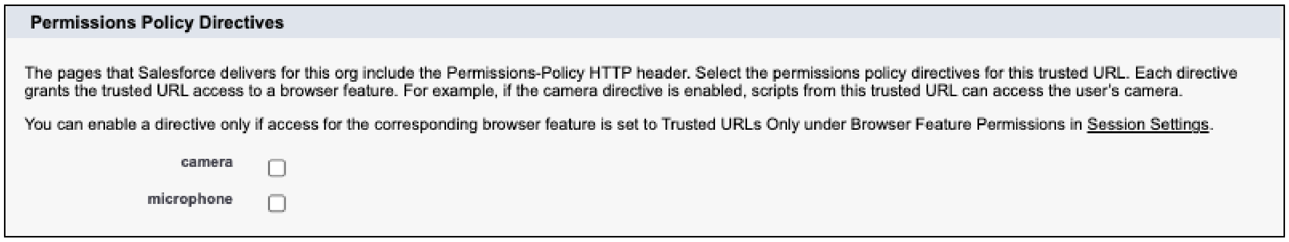
Grant a Trusted URL Access to Browser Features

Select the permissions policy directives for a trusted URL. Each directive grants the trusted URL access to a browser feature.

To use this feature, enable the Permissions-Policy header in Session Settings. You can control access to a browser feature at the trusted URL level only when access for the corresponding feature is set to Trusted URLs Only in Session Settings.

1. Add or edit a trusted URL.

   You grant access to browser features in the Permissions Policy Directives section of the Trusted URL page.



Permissions Policy Directives

The pages that Salesforce delivers for this org include the Permissions-Policy HTTP header. Select the permissions policy directives for this trusted URL. Each directive grants the trusted URL access to a browser feature. For example, if the camera directive is enabled, scripts from this trusted URL can access the user's camera.

You can enable a directive only if access for the corresponding browser feature is set to Trusted URLs Only under Browser Feature Permissions in Session Settings.

camera ☐

microphone ☐

2. To grant this trusted URL permission access to the user’s camera, select **camera**.
3. To grant this trusted URL permission access to the user’s camera, select **microphone**.

SEE ALSO:

Control Access to Browser Features

CHAPTER 11 Working with Salesforce Data

In this chapter ...

To create, read, and update Salesforce data from an Aura component, use Lightning Data Service via `force:recordData` or the form-based components. To delete Salesforce data, use `force:recordData`.

Lightning Data Service

Use Lightning Data Service to load, create, edit, or delete a record in your component without requiring Apex code. Lightning Data Service handles sharing rules and field-level security for you. In addition to simplifying access to Salesforce data, Lightning Data Service improves performance and user interface consistency.

At the simplest level, you can think of Lightning Data Service as the Lightning components version of the Visualforce standard controller. While this statement is an over-simplification, it serves to illustrate a point. Whenever possible, use Lightning Data Service to read and modify Salesforce data in your components.

Data access with Lightning Data Service is simpler than the equivalent using a server-side Apex controller. Read-only access can be entirely declarative in your component's markup. For code that modifies data, your component's JavaScript controller is roughly the same amount of code, and you eliminate the Apex entirely. All your data access code is consolidated into your component, which significantly reduces complexity.

Lightning Data Service provides other benefits aside from the code. It's built on highly efficient local storage that's shared across all components that use it. Records loaded in Lightning Data Service are cached and shared across components.

Note: Working with Lightning Data Service in Lightning Web Components? See the Lightning Web Components Developer Guide.

Components accessing the same record see significant performance improvements, because a record is loaded only once, no matter how many components are using it. Shared records also improve user interface consistency. When one component updates a record, the other components using it are notified, and in most cases, refresh automatically.

Creating Components That Use Lightning Data Service

Lightning Data Service is available through `force:recordData` and several base components. To return raw record data, for example if you need to view or edit only a few fields, and don't need any UI elements or layout information, use `force:recordData`. When using `force:recordData`, load the data once and pass it to child components as attributes. This approach reduces the number of listeners and minimizes server calls, which improves performance and ensures that your components show consistent data. For more information, see force:recordData documentation.

To create a form for working with records, use `lightning:recordForm`, `lightning:recordEditForm`, or `lightning:recordViewForm`. One advantage of using the form-based components is that you can achieve many of your record display needs entirely in markup without JavaScript. Another powerful feature of the form-based components is automatic field mapping with field-level validation. The form-based components use a base component that's appropriate for the field type to render the field automatically.

`force:recordData` doesn't include any UI elements; it's simply logic and a way to communicate to the server. Here are the components that use Lightning Data Service.

`lightning:recordForm`
Display, create, or edit records

`lightning:recordViewForm`
Display records with `lightning:outputField`

`lightning:recordEditForm`
Create or edit records with `lightning:inputField`

`force:recordData`
Create, edit, or delete record data using your own custom UI components

IN THIS SECTION:

Loading a Record

Loading a record can be accomplished entirely in markup using `lightning:recordForm`. If you need a custom layout, use `lightning:recordViewForm`. If you need more customization than the form-based components allow for viewing record data, use `force:recordData`.

Display a Record Using `lightning:recordForm`

To display a record using `lightning:recordForm`, provide the record ID and the object API name. Additionally, provide fields using either the `fields` or `layoutType` attribute. You can display a record in two modes using the `mode` attribute.

`view`
: Loads the form using output fields with inline editing enabled. Editable fields have edit icons. If a user clicks an edit icon, editable fields in the form become editable, and the form displays Cancel and Save buttons. This is the default mode when a record ID is provided.

`readonly`
: Loads the form with output fields only. The form doesn't include edit icons or Cancel and Save buttons.

This example displays an account record in view mode using the compact layout, which includes fewer fields than the full layout. The `columns` attribute displays the record fields in two columns that are evenly sized. Update the record ID with your own.

```
<aura:component>
    <lightning:recordForm
        recordId="001XXXXXXXXXXXXXXX"
        objectApiName="Account"
        layoutType="Compact"
        columns="2"/>
</aura:component>
```

To display the field values on a record page, implement the `flexipage:availableForRecordHome` and `flexipage:hasRecordId`. The component automatically inherits the record ID.

This example displays read-only values for the account's `Name` and `Industry` fields. Add this example to an account record page.

```
<aura:component implements="flexipage:availableForRecordHome,force:hasRecordId">
    <aura:attribute name="recordId" type="String" />
    <aura:attribute name="fields" type="String[]" default="['Name','Industry']" />
    <lightning:recordForm recordId="{!v.recordId}"
                          objectApiName="Account"
                          mode="readonly"
                          fields="{!v.fields}" />
```

If you provide both `fields` and `layoutType` attributes, the display order of the fields is not guaranteed. To specify the field order, use `fields` without the `layoutType` attribute. Alternatively, use the `lightning:recordViewForm` component as shown in the next section.

Display a Record with a Custom Layout Using `lightning:recordViewForm`

To display a read-only record with a custom layout, use the `lightning:recordViewForm` component. To compose a form field, use `lightning:outputField` components, which maps to a Salesforce field by using the `fieldName` attribute. Including individual fields lets you style a custom layout using the Lightning Design System utility classes, such as the grid system.

```
<aura:component>
    <lightning:recordViewForm recordId="001XXXXXXXXXXXXXXX"
                              objectApiName="Account">
    <div class="slds-grid">
        <div class="slds-col slds-size_2-of-3">
            <lightning:outputField fieldName="Name" />
            <lightning:outputField fieldName="Phone" />
        </div>
        <div class="slds-col slds-size_1-of-3">
```

```
            <lightning:outputField fieldName="Industry" />
            <lightning:outputField fieldName="AnnualRevenue" />
        </div>
    </div>
</lightning:recordViewForm>
</aura:component>
```

If you require more customization when displaying a record than what `lightning:recordForm` and `lightning:recordViewForm` allow, consider using `force:recordData`.

Display Record Data in a Custom User Interface Using `force:recordData`

`force:recordData` enables granular customization, including providing your own component to load data. To load a record using Lightning Data Service, add the `force:recordData` tag to your component and specify:

- The ID of the record to load
- A component attribute to assign the loaded record
- A list of fields to load

To specify a list of fields to load, use the `fields` attribute. For example, `fields="Name,BillingCity,BillingState"`.

Alternatively, you can specify a layout using the `layoutType` attribute. All fields on that layout are loaded for the record. The layout depends on the page layout assignment for the profile. For example, if a user using the Marketing User profile is assigned the default account layout, all fields on that layout are available to that user. Layouts are typically modified by administrators, so `layoutType` isn't as flexible as `fields` when you want to request specific fields. Loading record data using `layoutType` allows your component to adapt to layout definitions. Valid values for `layoutType` are `FULL` and `COMPACT`.

Note: We recommend that you use the `fields` attribute instead of `layoutType`. Use `layoutType` only if you want the administrator, not the component, to control the fields that are provisioned. The component must handle receiving every field that is assigned to the layout for the context user.

To get a field from an object regardless of whether an admin has included it in a layout, use the `fields` attribute and request the field by name.

`targetRecord` is populated with the current record, containing the fields relevant to the requested `layoutType` or the fields listed in the `fields` attribute. `targetFields` is populated with a simplified view of the loaded record. For example, for the `Name` field, `v.targetRecord.fields.Name.value` is equivalent to `v.targetFields.Name`.

Example: **Loading a Record**

The following example illustrates the essentials of loading a record using `force:recordData`. This component can be added to a record home page in the Lightning App Builder, or as a custom action. The record ID is supplied by the implicit `recordId` attribute added by the `force:hasRecordId` interface.

`ldsLoad.cmp`

```
<aura:component implements="flexipage:availableForRecordHome,
force:lightningQuickActionWithoutHeader, force:hasRecordId">

    <aura:attribute name="record" type="Object"/>
    <aura:attribute name="simpleRecord" type="Object"/>
    <aura:attribute name="recordError" type="String"/>

    <force:recordData aura:id="recordLoader"
      fields="Name,BillingCity,BillingState,Industry"
      recordId="{!v.recordId}"
      targetFields="{!v.simpleRecord}"
      targetError="{!v.recordError}"
      recordUpdated="{!c.handleRecordUpdated}"
      />

    <!-- Display a lightning card with details about the record -->
    <div class="Record Details">
    <lightning:card iconName="standard:account" title="{!v.simpleRecord.Name}" >
        <div class="slds-p-horizontal--small">
            <p class="slds-text-heading--small">
                <lightning:formattedText title="Billing City"
value="{!v.simpleRecord.BillingCity}" /></p>
            <p class="slds-text-heading--small">
                <lightning:formattedText title="Billing State"
value="{!v.simpleRecord.BillingState}" /></p>
        </div>
    </lightning:card>
    </div>

    <!-- Display Lightning Data Service errors, if any -->
    <aura:if isTrue="{!not(empty(v.recordError))}">
        <div class="recordError">
            {!v.recordError}</div>
    </aura:if>
</aura:component>
```

When you use the `fields` attribute, the `targetFields` attribute returns the record's `Id` and `SystemModstamp` fields, in addition to the fields you requested. In this example, `{!v.simpleRecord}` returns:

```
{
  "Id":"0011a0000000000000",
  "Name":"Salesforce",
  "SystemModstamp":"2020-06-14T23:44:43.000Z",
  "BillingCity":"San Franscisco",
  "BillingState":"CA",
  "Industry":"Technology"
}
```

`ldsLoadController.js`

```
({
    handleRecordUpdated: function(component, event, helper) {
        var eventParams = event.getParams();
        if(eventParams.changeType === "LOADED") {
           // record is loaded (render other component which needs record data value)

            console.log("Record is loaded successfully.");
            console.log("You loaded a record in " +
                        component.get("v.simpleRecord.Industry"));
        } else if(eventParams.changeType === "CHANGED") {
            // record is changed
        } else if(eventParams.changeType === "REMOVED") {
            // record is deleted
        } else if(eventParams.changeType === "ERROR") {
            // there's an error while loading, saving, or deleting the record
        }
    }
})
```

When the record loads or updates, to access the record fields in the JavaScript controller, use the `component.get("v.simpleRecord.fieldName")` syntax.

`force:recordData` loads data asynchronously by design since it may go to the server to retrieve data. To track when the record is loaded or changed, use the `recordUpdated` event as shown in the previous example. Alternatively, you can place a change handler on the attribute provided to `targetRecord` or `targetFields`.

SEE ALSO:

Component Library: `lightning:recordForm`

Component Library: `lightning:recordViewForm`

Configure Components for Lightning Experience Record Pages

Configure Components for Record-Specific Actions

Editing a Record

The simplest way to create a form that enables you to edit a record is to use the `lightning:recordForm` component. If you want to customize the form layout or preload custom values, use `lightning:recordEditForm`. If you want to customize a form more than the form-based components allow, use `force:recordData`.

Edit a Record using `lightning:recordForm`

To edit a record using `lightning:recordForm`, provide the record ID and object API name. When you provide a record ID, view mode is the default mode of this component, which displays fields with edit icons. If you click an edit icon, all fields in the form become editable.

This example creates a form that lets users update fields on an account record when an edit icon is clicked. It displays the fields from the compact layout in two columns. Add this example component to an account record page. The component inherits the record ID via the `force:hasRecordId` interface.

```
<aura:component implements="flexipage:availableForRecordHome, force:hasRecordId">
    <lightning:recordForm
        recordId = "{!v.recordId}"
        objectApiName="Account"
        layoutType="Compact"
        columns="2" />
</aura:component>
```

When the record is saved successfully, all components that contain the updated field values are refreshed automatically.

Add `mode="edit"` to transform the form to one that displays input fields for editing. The form displays a Save button that updates the record, and a Cancel button that reverts changes.

```
<aura:component implements="flexipage:availableForRecordHome, force:hasRecordId">
    <lightning:recordForm
        recordId = "{!v.recordId}"
        objectApiName="Account"
        layoutType="Compact"
        mode="edit" />
</aura:component>
```

Customize Error Handling in **`lightning:recordForm`**

To customize the behavior when a record is saved successfully, use the `onsuccess` event handler. Errors are automatically handled and displayed. To customize them, use the `onerror` event handler.

```
<aura:component implements="flexipage:availableForRecordHome, force:hasRecordId">

    <!-- Displays a toast notification -->
    <lightning:notificationsLibrary aura:id="notifLib" />
    <lightning:recordForm
        recordId = "{!v.recordId}"
        objectApiName="Account"
        layoutType="Compact"
        mode="edit"
        onsuccess="{!c.handleSuccess}"
        onerror="{!c.handleError}"/>
</aura:component>
```

A toast notification is displayed when a record is saved successfully or when an error is encountered during save.

```
({
    handleSuccess: function (cmp, event, helper) {
        cmp.find('notifLib').showToast({
            "title": "Record updated!",
            "message": "The record "+ event.getParam("id") + " has been updated
successfully.",
            "variant": "success"
        });
    },

    handleError: function (cmp, event, helper) {
        cmp.find('notifLib').showToast({
            "title": "Something has gone wrong!",
            "message": event.getParam("message"),
            "variant": "error"
        });
    }
})
```

Note: For more information, see lightning:recordForm.

Edit a Record with a Custom Layout Using **`lightning:recordEditForm`**

To provide a custom layout for your form fields, use the `lightning:recordEditForm` component.

Pass in the fields to `lightning:inputField`, which displays an input control based on the record field type.

This example displays a form with two fields using a custom layout. Add this example component to an account record page.

```
<aura:component implements="flexipage:availableForRecordHome, force:hasRecordId">
    <lightning:recordEditForm
```

```
        recordId="{!v.recordId}"
        objectApiName="Account">
        <lightning:messages />
            <div class="slds-grid">
                <div class="slds-col slds-size_1-of-2">
                    <lightning:inputField fieldName="Name"/>
                </div>
                <div class="slds-col slds-size_1-of-2">
                    <lightning:inputField fieldName="Industry"/>
                </div>
            </div>
        <lightning:button class="slds-m-top_small" type="submit" label="Create new" />
    </lightning:recordEditForm>
</aura:component>
```

When a server error is encountered, `lightning:recordEditForm` displays an error message above the form fields using the `lightning:messages` component. Alternatively, provide your own error handling using the `onerror` event handler.

Another feature that `lightning:recordEditForm` provides that's not available with `lightning:recordForm` is displaying the form with custom field values, as shown in the next section.

Prepopulate Field Values

To provide a custom field value when the form displays, use the `value` attribute on `lightning:inputField`. If you're providing a record ID, the value returned by the record on load does not override this custom value.

Alternatively, set the field value using this syntax.

```
cmp.find("nameField").set("v.value", "My New Account Name");
```

Note: For more information, see lightning:recordEditForm.

If you require more customization when creating a record than what `lightning:recordForm` and `lightning:recordEditForm` allow, consider using `force:recordData`.

Edit a Record via a Custom User Interface Using `force:recordData`

To edit and save a record using `force:recordData`, call `saveRecord` and pass in a callback function to be invoked after the save operation completes. The save operation is used in two cases.

- To save changes to an existing record
- To create and save a new record

To save changes to an existing record, load the record in EDIT mode and call `saveRecord` on the `force:recordData` component.

To save a new record, and thus create it, create the record from a record template, as described in Creating a Record. Then call `saveRecord` on the `force:recordData` component.

Load a Record in EDIT Mode

To load a record that might be updated, set the `force:recordData` tag's `mode` attribute to "EDIT". Other than explicitly setting the `mode`, loading a record for editing is the same as loading it for any other purpose.

Note: Since Lightning Data Service records are shared across multiple components, loading records provides the component with a copy of the record instead of a direct reference. If a component loads a record in VIEW mode, Lightning Data Service automatically overwrites that copy with a newer copy of the record when the record is changed. If a record is loaded in EDIT mode, the record is not updated when the record is changed. This prevents unsaved changes from appearing in components that reference the record while the record is being edited, and prevents any edits in progress from being overwritten. Notifications are still sent in both modes.

Call `saveRecord` to Save Record Changes

To perform the save operation, call `saveRecord` on the `force:recordData` component from the appropriate controller action handler. The `saveRecord` method takes one argument—a callback function to be invoked when the operation completes. This callback function receives a `SaveRecordResult` as its only parameter. `SaveRecordResult` includes a `state` attribute that indicates success or error, and other details you can use to handle the result of the operation.

Example: **Saving a Record**

The following example illustrates the essentials of saving a record using Lightning Data Service. It's intended for use on a record page. The record ID is supplied by the implicit `recordId` attribute added by the `force:hasRecordId` interface.

`ldsSave.cmp`

```
<aura:component implements="flexipage:availableForRecordHome,force:hasRecordId">

    <aura:attribute name="record" type="Object"/>
    <aura:attribute name="simpleRecord" type="Object"/>
    <aura:attribute name="recordError" type="String"/>

    <force:recordData aura:id="recordHandler"
      recordId="{!v.recordId}"
      fields="Name,BillingState,BillingCity"
      targetRecord="{!v.record}"
      targetFields="{!v.simpleRecord}"
      targetError="{!v.recordError}"
      mode="EDIT"
      recordUpdated="{!c.handleRecordUpdated}"
      />

    <!-- Display a lightning card with details about the record -->
    <div class="Record Details">
        <lightning:card iconName="standard:account" title="{!v.simpleRecord.Name}" >
            <div class="slds-p-horizontal--small">
                <p class="slds-text-heading--small">
                    <lightning:formattedText title="Billing State"
value="{!v.simpleRecord.BillingState}" /></p>
                <p class="slds-text-heading--small">
                    <lightning:formattedText title="Billing City"
value="{!v.simpleRecord.BillingCity}" /></p>
            </div>
        </lightning:card>
    </div>

    <!-- Display an editing form -->
    <div class="Record Details">
        <lightning:card iconName="action:edit" title="Edit Account">
            <div class="slds-p-horizontal--small">
               <lightning:input label="Account Name" value="{!v.simpleRecord.Name}"/>
```

```
                <br/>
                <lightning:button label="Save Account" variant="brand"
onclick="{!c.handleSaveRecord}" />
            </div>
        </lightning:card>
    </div>

    <!-- Display Lightning Data Service errors, if any -->
    <aura:if isTrue="{!not(empty(v.recordError))}">
        <div class="recordError">
            {!v.recordError}</div>
    </aura:if>
</aura:component>
```

To improve performance, we recommend using the `fields` attribute to query only the fields you need. Use `layoutType` only if you expect to display or edit a large number of fields on the compact or full layout.

Note: To edit the constituent fields on compound fields, such as the FirstName and LastName fields in the Name compound field, create a separate `lightning:input` component for `{!v.simpleRecord.FirstName}` and `{!v.simpleRecord.LastName}`.

This component loads a record using `force:recordData` set to EDIT mode, and provides a form for editing record values. (In this simple example, just the record name field.)

`ldsSaveController.js`

```
({
    handleSaveRecord: function(component, event, helper) {
       component.find("recordHandler").saveRecord($A.getCallback(function(saveResult)
 {
            // use the recordUpdated event handler to handle generic logic when record
 is changed
            if (saveResult.state === "SUCCESS" || saveResult.state === "DRAFT") {
                // handle component related logic in event handler
            } else if (saveResult.state === "INCOMPLETE") {
                console.log("User is offline, device doesn't support drafts.");
            } else if (saveResult.state === "ERROR") {
                console.log('Problem saving record, error: ' +
JSON.stringify(saveResult.error));
            } else {
                console.log('Unknown problem, state: ' + saveResult.state + ', error:
 ' + JSON.stringify(saveResult.error));
            }
        }));
    },

    /**
     * Control the component behavior here when record is changed (via any component)

     */
    handleRecordUpdated: function(component, event, helper) {
        var eventParams = event.getParams();
        if(eventParams.changeType === "CHANGED") {
            // get the fields that changed for this record
            var changedFields = eventParams.changedFields;
            console.log('Fields that are changed: ' + JSON.stringify(changedFields));

            // record is changed, so refresh the component (or other component logic)

            var resultsToast = $A.get("e.force:showToast");
            resultsToast.setParams({
                "title": "Saved",
                "message": "The record was updated."
            });
            resultsToast.fire();

        } else if(eventParams.changeType === "LOADED") {
            // record is loaded in the cache
        } else if(eventParams.changeType === "REMOVED") {
            // record is deleted and removed from the cache
        } else if(eventParams.changeType === "ERROR") {
            // there's an error while loading, saving or deleting the record
        }
    }
})
```

The `handleSaveRecord` action here is a minimal version. There's no form validation or real error handling. Whatever is entered in the form is attempted to be saved to the record.

If you are creating multiple instances of `force:recordData` on a page, provide your `saveRecord` and `recordUpdated` handlers accordingly. For example, if you have two instances of `force:recordData` that updates the same record, assign a different `aura:id` to each instance, such that `saveRecord` is called uniquely, and subsequently the `recordUpdated` handler.

SEE ALSO:

Component Library: `lightning:recordForm`

Component Library: `lightning:recordEditForm`

SaveRecordResult

Configure Components for Lightning Experience Record Pages

Configure Components for Record-Specific Actions

Creating a Record

The simplest way to create a form that enables users create a record is to use `lightning:recordForm`. If you want to customize the form layout or preload custom values, use `lightning:recordEditForm`. If you need more customization than the form-based components allow, use `force:recordData`.

Create a Record using `lightning:recordForm`

To create a record using `lightning:recordForm`, leave out the `recordId` attribute.

This example displays a form that creates an account record with a list of fields. The Cancel and Save buttons are displayed at the bottom of the form.

```
<aura:component>
    <aura:attribute name="fields"
                    type="String[]"
                    default="['Name', 'Industry']"/>
    <lightning:recordForm objectApiName="Account"
                          fields="{!v.fields}"/>
</aura:component>
```

When the record saves successfully, the fields display pencil icons to denote that inline editing is available. This view is displayed until you refresh or reload the page. Then the form redisplays the record fields without data, ready to create a new record.

Alternatively, use the `Full` layout type, which loads all fields from the full layout to display a form that creates a record. The `columns` attribute displays the record fields in two columns that are evenly sized.

```
<aura:component>
    <lightning:recordForm objectApiName="Account"
                          layoutType="Full"
                          columns="2"/>
</aura:component>
```

Customize Error Handling in `lightning:recordForm`

When an error is encountered during save, `lightning:recordForm` displays an error message at the top of the form. You can provide additional error handling using the `onerror` event handler.

This example displays a toast message when an error is returned.

```
<aura:component>
    <aura:attribute name="fields"
                    type="String[]"
                    default="['Name', 'Industry']"/>

    <!-- Displays toast notifications -->
    <lightning:notificationsLibrary aura:id="notifLib" />
    <lightning:recordForm
        objectApiName="Account"
        fields="{!v.fields}"
        onerror="{!c.handleError}"/>
</aura:component>
```

To return the error message, use `event.getParam("message")`.

```
({
    handleError: function (cmp, event, helper) {
        cmp.find('notifLib').showToast({
            "title": "Something has gone wrong!",
            "message": event.getParam("message"),
            "variant": "error"
        });
    }
})
```

To customize the form behavior when a record saves successfully, use the `onsuccess` event handler.

If you want to provide a custom layout or load custom field values when the form displays, use the `lightning:recordEditForm` component as shown in the next section.

Create a Record with a Custom Layout Using `lightning:recordEditForm`

To provide a custom layout for your form fields, use the `lightning:recordEditForm` component.

Pass in the fields to `lightning:inputField`, which displays an input control based on the record field type.

This example creates a custom layout using the Grid utility classes in Lightning Design System.

```
<aura:component>
    <lightning:recordEditForm objectApiName="Account">
        <lightning:messages />
            <div class="slds-grid">
                <div class="slds-col slds-size_2-of-3">
                    <lightning:inputField fieldName="Name"/>
                </div>
                <div class="slds-col slds-size_1-of-3">
                    <lightning:inputField fieldName="Industry"/>
                </div>
            </div>
        <lightning:button class="slds-m-top_small" type="submit" label="Create new" />
    </lightning:recordEditForm>
</aura:component>
```

When a server error is encountered,`lightning:recordEditForm` displays an error message above the form fields. To enable automatic error handling, include the `lightning:messages` component. Alternatively, provide your own error handling using the `onerror` event handler.

Another feature that `lightning:recordEditForm` provides that's not available with `lightning:recordForm` is preset custom field values, as shown in the next section.

Prepopulate Field Values

To provide a custom field value when the form displays, use the `value` attribute on `lightning:inputField`. If you're providing a record ID, the value returned by the record on load does not override this custom value.

Alternatively, set the field value using this syntax.

```
cmp.find("nameField").set("v.value", "My New Account Name");
```

Note: For more information, see lightning:recordEditForm.

If you require more customization when creating a record than what `lightning:recordForm` and `lightning:recordEditForm` allow, consider using `force:recordData`.

Create a Record via a Custom User Interface Using `force:recordData`

To create a record using `force:recordData`, leave out the `recordId` attribute. Load a record template by calling the `getNewRecord` function on `force:recordData`. Finally, apply values to the new record, and save the record by calling the `saveRecord` function on `force:recordData`.

1. Call `getNewRecord` to create an empty record from a record template. You can use this record as the backing store for a form or otherwise have its values set to data intended to be saved.

2. Call `saveRecord` to commit the record. This is described in Editing a Record.

Create an Empty Record from a Record Template

To create an empty record from a record template, you can't set a `recordId` on the `force:recordData` tag. Without a `recordId`, Lightning Data Service doesn't load an existing record.

In your component's `init` or another handler, call the `getNewRecord` on `force:recordData`. `getNewRecord` takes the following arguments.

Attribute Name	Type	Description
`objectApiName`	String	The object API name for the new record.
`recordTypeId`	String	The 18 character ID of the record type for the new record. If not specified, the default record type for the object is used, as defined in the user's profile.
`skipCache`	Boolean	Whether to load the record template from the server instead of the client-side Lightning Data Service cache. Defaults to `false`.
`callback`	Function	A function invoked after the empty record is created. This function receives no arguments.

`getNewRecord` doesn't return a result. It simply prepares an empty record and assigns it to the `targetRecord` attribute.

Example: Creating a Record

The following example illustrates the essentials of creating a record using Lightning Data Service. This example is intended to be added to an account record Lightning page.

`ldsCreate.cmp`

```
<aura:component implements="flexipage:availableForRecordHome, force:hasRecordId">

    <aura:attribute name="newContact" type="Object"/>
    <aura:attribute name="simpleNewContact" type="Object"/>
    <aura:attribute name="newContactError" type="String"/>

    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <force:recordData aura:id="contactRecordCreator"
                      fields="FirstName,LastName,Title"
                      targetRecord="{!v.newContact}"
                      targetFields="{!v.simpleNewContact}"
                      targetError="{!v.newContactError}" />

    <!-- Display the new contact form -->
    <div class="Create Contact">
        <lightning:card iconName="action:new_contact" title="Create Contact">
            <div class="slds-p-horizontal--small">
                <lightning:input aura:id="contactField" label="First Name"
value="{!v.simpleNewContact.FirstName}"/>
                <lightning:input aura:id="contactField" label="Last Name"
```

```
value="{!v.simpleNewContact.LastName}"/>
                <lightning:input aura:id="contactField" label="Title"
value="{!v.simpleNewContact.Title}"/>
                <br/>
                <lightning:button label="Save Contact" variant="brand"
onclick="{!c.handleSaveContact}"/>
            </div>
        </lightning:card>
    </div>

    <!-- Display Lightning Data Service errors -->
    <aura:if isTrue="{!not(empty(v.newContactError))}">
        <div class="recordError">
            {!v.newContactError}</div>
    </aura:if>

</aura:component>
```

Note: To improve performance, we recommend using the `fields` attribute to query only the fields you need. Use `layoutType` only if you want the administrator, not the component, to control the fields that are provisioned. The component must handle receiving every field that is assigned to the layout for the context user.

This component doesn't set the `recordId` attribute of `force:recordData`. This tells Lightning Data Service to expect a new record. Here, that's created in the component's `init` handler.

`ldsCreateController.js`

```
({
    doInit: function(component, event, helper) {
        // Prepare a new record from template
        component.find("contactRecordCreator").getNewRecord(
            "Contact", // sObject type (objectApiName)
            null,      // recordTypeId
            false,     // skip cache?
            $A.getCallback(function() {
                var rec = component.get("v.newContact");
                var error = component.get("v.newContactError");
                if(error || (rec === null)) {
                    console.log("Error initializing record template: " + error);
                    return;
                }
                console.log("Record template initialized: " + rec.apiName);
            })
        );
    },

    handleSaveContact: function(component, event, helper) {
        if(helper.validateContactForm(component)) {
          component.set("v.simpleNewContact.AccountId", component.get("v.recordId"));

            component.find("contactRecordCreator").saveRecord(function(saveResult) {
                if (saveResult.state === "SUCCESS" || saveResult.state === "DRAFT") {

                    // record is saved successfully
                    var resultsToast = $A.get("e.force:showToast");
                    resultsToast.setParams({
                        "title": "Saved",
                        "message": "The record was saved."
                    });
                    resultsToast.fire();

                } else if (saveResult.state === "INCOMPLETE") {
                    // handle the incomplete state
                    console.log("User is offline, device doesn't support drafts.");
                } else if (saveResult.state === "ERROR") {
                    // handle the error state
                    console.log('Problem saving contact, error: ' +
JSON.stringify(saveResult.error));
                } else {
                    console.log('Unknown problem, state: ' + saveResult.state + ',
error: ' + JSON.stringify(saveResult.error));
                }
            });
        }
    }
})
```

The `doInit` init handler calls `getNewRecord()` on the `force:recordData` component, passing in a simple callback handler. This call returns a Record object to create an empty contact record, which is used by the contact form in the component's markup.

Note: The callback passed to `getNewRecord()` must be wrapped in `$A.getCallback()` to ensure correct access context when the callback is invoked. If the callback is passed in without being wrapped in `$A.getCallback()`, any attempt to access private attributes of your component results in access check failures.

Even if you're not accessing private attributes, it's a best practice to always wrap the callback function for `getNewRecord()` in `$A.getCallback()`. Never mix (contexts), never worry.

The `handleSaveContact` handler is called when the **Save Contact** button is clicked. It's a straightforward application of saving the contact, as described in Editing a Record, and then updating the user interface.

Note: The helper function, `validateContactForm`, isn't shown. It simply validates the form values. For an example of this validation, see Lightning Action Examples.

SEE ALSO:

Component Library: `lightning:recordForm`

Component Library: `lightning:recordEditForm`

Editing a Record

Configure Components for Lightning Experience Record Pages

Configure Components for Record-Specific Actions

Controlling Access

Deleting a Record

To delete a record using Lightning Data Service, call `deleteRecord` on the `force:recordData` component, and pass in a callback function to be invoked after the delete operation completes. The form-based components, such as `lightning:recordForm`, don't currently support deleting a record.

Delete operations with Lightning Data Service are straightforward. The `force:recordData` tag can include minimal details. If you don't need any record data, set the `fields` attribute to just `Id`. If you know that the only operation is a delete, any `mode` can be used.

To perform the delete operation, call `deleteRecord` on the `force:recordData` component from the appropriate controller action handler. `deleteRecord` takes one argument, a callback function to be invoked when the operation completes. This callback function receives a `SaveRecordResult` as its only parameter. `SaveRecordResult` includes a `state` attribute that indicates success or error, and other details you can use to handle the result of the operation.

Example: **Deleting a Record**

The following example illustrates the essentials of deleting a record using Lightning Data Service. This component adds a **Delete Record** button to a record page, which deletes the record being displayed. The record ID is supplied by the implicit `recordId` attribute added by the `force:hasRecordId` interface.

`ldsDelete.cmp`

```
<aura:component implements="flexipage:availableForRecordHome,force:hasRecordId">

    <aura:attribute name="recordError" type="String" access="private"/>

    <force:recordData aura:id="recordHandler"
        recordId="{!v.recordId}"
        fields="Id"
        targetError="{!v.recordError}"
        recordUpdated="{!c.handleRecordUpdated}" />

    <!-- Display the delete record form -->
    <div class="Delete Record">
        <lightning:card iconName="action:delete" title="Delete Record">
            <div class="slds-p-horizontal--small">
                <lightning:button label="Delete Record" variant="destructive"
onclick="{!c.handleDeleteRecord}"/>
            </div>
        </lightning:card>
    </div>

    <!-- Display Lightning Data Service errors, if any -->
    <aura:if isTrue="{!not(empty(v.recordError))}">
        <div class="recordError">
            {!v.recordError}</div>
    </aura:if>
</aura:component>
```

Notice that the `force:recordData` tag includes only the `recordId` and a nearly empty `fields` list—the absolute minimum required. If you want to display record values in the user interface, for example, as part of a confirmation message, define the `force:recordData` tag as you would for a load operation instead of this minimal delete example.

ldsDeleteController.js

```
({
    handleDeleteRecord: function(component, event, helper) {

component.find("recordHandler").deleteRecord($A.getCallback(function(deleteResult) {
            // NOTE: If you want a specific behavior(an action or UI behavior) when
this action is successful
            // then handle that in a callback (generic logic when record is changed
should be handled in recordUpdated event handler)
            if (deleteResult.state === "SUCCESS" || deleteResult.state === "DRAFT") {

                // record is deleted
                console.log("Record is deleted.");
            } else if (deleteResult.state === "INCOMPLETE") {
                console.log("User is offline, device doesn't support drafts.");
            } else if (deleteResult.state === "ERROR") {
                console.log('Problem deleting record, error: ' +
JSON.stringify(deleteResult.error));
            } else {
               console.log('Unknown problem, state: ' + deleteResult.state + ', error:
 ' + JSON.stringify(deleteResult.error));
            }
        }));
    },

    /**
     * Control the component behavior here when record is changed (via any component)

     */
    handleRecordUpdated: function(component, event, helper) {
        var eventParams = event.getParams();
        if(eventParams.changeType === "CHANGED") {
           // record is changed
        } else if(eventParams.changeType === "LOADED") {
            // record is loaded in the cache
        } else if(eventParams.changeType === "REMOVED") {
            // record is deleted, show a toast UI message
            var resultsToast = $A.get("e.force:showToast");
            resultsToast.setParams({
                "title": "Deleted",
                "message": "The record was deleted."
            });
            resultsToast.fire();

        } else if(eventParams.changeType === "ERROR") {
            // there's an error while loading, saving, or deleting the record
        }
    }
})
```

When the record is deleted, navigate away from the record page. Otherwise, you see a "record not found" error when the component refreshes. Here the controller uses the `objectApiName` property in the `SaveRecordResult` provided to the callback function, and navigates to the object home page.

SEE ALSO:

SaveRecordResult

Configure Components for Lightning Experience Record Pages

Configure Components for Record-Specific Actions

Record Changes

To perform more advanced tasks using `force:recordData` when the record changes, handle the `recordUpdated` event. You can handle record loaded, updated, and deleted changes, applying different actions to each change type.

If a component performs logic that's specific to the record data, it must run that logic again when the record changes. A common example is a business process in which the actions that apply to a record change depending on the record's values. For example, different actions apply to opportunities at different stages of the sales cycle.

Note: Lightning Data Service notifies listeners about data changes only if the changed fields are the same as in the listener's fields or layout.

Example: Declare that your component handles the `recordUpdated` event. To improve performance, we recommend using the `fields` attribute to query only the fields you need. Use `layoutType` only if you want the administrator, not the component, to control the fields that are provisioned. The component must handle receiving every field that is assigned to the layout for the context user.

```
<force:recordData aura:id="forceRecord"
    recordId="{!v.recordId}"
    fields="Name,Title,Email"
    targetRecord="{!v._record}"
    targetFields="{!v.simpleRecord}"
    targetError="{!v._error}"
    recordUpdated="{!c.recordUpdated}" />
```

Implement an action handler that handles the change.

```
({
  recordUpdated: function(component, event, helper) {

    var changeType = event.getParams().changeType;

    if (changeType === "ERROR") { /* handle error; do this first! */ }
    else if (changeType === "LOADED") { /* handle record load */ }
    else if (changeType === "REMOVED") { /* handle record removal */ }
    else if (changeType === "CHANGED") { /* handle record change */ }
})
```

When loading a record in edit mode, the record isn't automatically updated to prevent edits currently in progress from being overwritten. To update the record, use the `reloadRecord` method in the action handler.

```
<force:recordData aura:id="forceRecord"
    recordId="{!v.recordId}"
    fields="Name,Title,Email"
    targetRecord="{!v._record}"
    targetFields="{!v.simpleRecord}"
    targetError="{!v._error}"
    mode="EDIT"
    recordUpdated="{!c.recordUpdated}" />
```

```
({
  recordUpdated : function(component, event, helper) {

    var changeType = event.getParams().changeType;

    if (changeType === "ERROR") { /* handle error; do this first! */ }
    else if (changeType === "LOADED") { /* handle record load */ }
    else if (changeType === "REMOVED") { /* handle record removal */ }
    else if (changeType === "CHANGED") {
      /* handle record change; reloadRecord will cause you to lose your current record,
 including any changes you've made */
      component.find("forceRecord").reloadRecord();}
    }
})
```

Handling Errors

Lightning Data Service returns an error when a resource, such as a record or an object, is inaccessible on the server.

For example, an error occurs if you pass in an invalid input to the form-based components, such as an invalid record ID or missing required fields. An error is also returned if the record isn't in the cache and the server is offline. Also, a resource can become inaccessible on the server when it's deleted or has its sharing or visibility settings updated.

Handle Errors For Form-Based Components

Two types of errors—field-level errors and Lightning Data Service errors—are handled by `lightning:recordForm`, `lightning:recordEditForm`, and `lightning:recordViewForm`. Field-validation errors display below a field and cannot be customized. For example, an error is shown below a required field when it's empty. Lightning Data Service errors are handled in the following ways.

`lightning:recordForm`
: Automatically displays an error message above the form fields. You can provide additional error handling using the `onerror` event handler.

`lightning:recordEditForm`
: To automatically display an error message above or below the form fields, include `lightning:messages` before or after your `lightning:inputField` components.

  You can provide additional error handling using the `onerror` event handler.

`lightning:recordViewForm`

To automatically display an error message above or below the form fields, include `lightning:messages` before or after your `lightning:outputField` components.

If a single field has multiple validation errors, the form shows only the first error on the field. Similarly, if a submitted form has multiple errors, the form displays only the first error encountered. When you correct the displayed error, the next error is displayed.

The error object looks like this.

```
{
  "message": "Disconnected or Canceled",
  "detail": "",
  "output": {

  },
  "error": {
    "ok": false,
    "status": 400,
    "statusText": "Bad Request",
    "body": {
      "message": "Disconnected or Canceled"
    }
  }
}
```

Get the error object using this syntax.

```
handleError: function (cmp, event, helper) {
    var error = event.getParams();

    // Get the error message
    var errorMessage = event.getParam("message");
}
```

Handle Errors For `force:recordData`

To act when an error occurs, handle the `recordUpdated` event and handle the case where the `changeType` is "ERROR".

Example: Declare that your component handles the `recordUpdated` event.

```
<force:recordData aura:id="forceRecord"
    recordId="{!v.recordId}"
    fields="Name,Title,Email"
    targetRecord="{!v._record}"
    targetFields="{!v.simpleRecord}"
    targetError="{!v._error}"
    recordUpdated="{!c.recordUpdated}" />
```

Implement an action handler that handles the error.

```
({
  recordUpdated: function(component, event, helper) {

    var changeType = event.getParams().changeType;
    if (changeType === "ERROR") { /* handle error; do this first! */ }
    else if (changeType === "LOADED") { /* handle record load */ }
    else if (changeType === "REMOVED") { /* handle record removal */ }
    else if (changeType === "CHANGED") { /* handle record change */ }
})
```

If an error occurs when the record begins to load, `targetError` is set to a localized error message. An error occurs if:

- Input is invalid because of an invalid attribute value, or combination of attribute values. For example, an invalid `recordId`, or omitting both the `layoutType` and the `fields` attributes.
- The record isn't in the cache and the server is unreachable (offline).

If the record becomes inaccessible on the server, the `recordUpdated` event is fired with `changeType` set to "REMOVED." No error is set on `targetError`, since records becoming inaccessible is sometimes the expected outcome of an operation. For example, after lead convert the lead record becomes inaccessible.

Records can become inaccessible for the following reasons.

- Record or entity sharing or visibility settings restrict access.
- Record or entity is deleted.

When the record becomes inaccessible on the server, the record's JavaScript object assigned to `targetRecord` is unchanged.

Changing the Display Density

In Lightning Experience, the display density setting determines how densely content is displayed and where field labels are located. Display density is controlled for the org in Setup, and users can also set display density to their liking from their profile menu.

An org's comfy setting places the labels on the top of fields and adds more space between page elements. Contrastingly, compact is a denser view with labels on the same line as the fields and less space between lines. The cozy setting resembles compact, but with more space between lines.

You can design your form to respect the display density setting, or set the form density to override the display density setting. Overriding display density gives you more control over the label location, but doesn't affect spacing. In addition, you can set individual fields in your form to use variants that change the label location for the field.

Use the Org's Default Display Density in a Form

`lightning:recordEditForm`, `lightning:recordViewForm`, and `lightning:recordForm` adapt to your org's display density by default or when you set `density="auto"`.

```
<lightning:card iconName="standard:contact" title="recordEditForm">
    <div class="slds-p-horizontal_small">
        <!-- Replace the recordId with your own -->
        <lightning:recordEditForm recordId="003RM0000066Y82YAE"
                                  objectApiName="Contact"
                                  density="auto">
            <lightning:messages />
            <lightning:inputField fieldName="FirstName" />
            <lightning:inputField fieldName="LastName" />
            <lightning:inputField fieldName="Email" />
            <lightning:inputField fieldName="Phone" />
        </lightning:recordEditForm>
    </div>
</lightning:card>
```

Override the Org's Display Density

To override the org's display density, specify `density="compact"` or `density="comfy"`. The `cozy` value isn't supported on the `density` attribute. If an org's display density is set to cozy, labels and fields are on the same line by default.

The following table lists the org's display density settings and how they relate to the form density on `lightning:recordEditForm`, `lightning:recordViewForm`, and `lightning:recordForm`.

Org Display Density	Form Density	Field Label Alignment
Comfy	`auto` (default) or `comfy`	Labels are above fields
	`compact`	Labels and fields are on the same line
Cozy	`auto` (default) or `compact`	Labels and fields are on the same line
	`comfy`	Labels are above fields
Compact	`auto` (default) or `compact`	Labels and fields are on the same line
	`comfy`	Labels are above fields

Reduce Space Between the Label and Field

When the form density is `compact`, the labels and fields can appear too far apart for a single column form in a larger region. To reduce the space between the label and field when the form uses compact density, use the `slds-form-element_1-col` class on `lightning:inputField` or `lightning:outputField`.

```
<lightning:card iconName="standard:contact" title="recordEditForm">
    <div class="slds-p-horizontal_small">
        <!-- Replace the recordId with your own -->
        <lightning:recordEditForm recordId="003RM0000066Y82YAE"
                                  objectApiName="Contact"
                                  density="compact">
            <lightning:messages />
            <lightning:inputField fieldName="FirstName" class="slds-form-element_1-col"/>

            <lightning:inputField fieldName="LastName" class="slds-form-element_1-col"/>
            <lightning:inputField fieldName="Email" class="slds-form-element_1-col"/>
            <lightning:inputField fieldName="Phone" class="slds-form-element_1-col"/>

        </lightning:recordEditForm>
    </div>
</lightning:card>
```

Set Label Variants on Form Fields

You can set a variant on `lightning:inputField` if you want specific fields to have a label and field alignment that's different than that used by the form. A variant overrides the display density for that field.

`lightning:inputField` supports these variants: `standard` (default), `label-hidden`, `label-inline`, and `label-stacked`.

This example displays two input fields with inline labels, while the rest of the fields have labels displayed on top of fields due to the comfy form density.

```
<lightning:card iconName="standard:contact" title="recordEditForm">
    <div class="slds-p-horizontal_small">
        <!-- Replace the recordId with your own -->
        <lightning:recordEditForm recordId="003RM0000066Y82YAE"
                                  objectApiName="Contact"
                                  density="comfy">
            <lightning:messages/>
            <lightning:inputField fieldName="FirstName" variant="label-inline"/>
            <lightning:inputField fieldName="LastName" variant="label-inline"/>
            <lightning:inputField fieldName="Email"/>
            <lightning:inputField fieldName="Phone"/>
        </lightning:recordEditForm>
    </div>
</lightning:card>
```

`lightning:outputField` supports these variants: `standard` (default) and `label-hidden`.

This example displays output field values without labels when the form density is `comfy`. Hidden labels are available to assistive technology.

```
<lightning:card iconName="standard:contact" title="recordViewForm">
    <div class="slds-p-horizontal_small">
        <!-- Replace the recordId with your own -->
        <lightning:recordViewForm recordId="003RM0000066Y82YAE"
                                  objectApiName="Contact"
                                  density="comfy">
            <lightning:messages />
            <lightning:outputField fieldName="FirstName" variant="label-hidden"/>
            <lightning:outputField fieldName="LastName" variant="label-hidden"/>
            <lightning:outputField fieldName="Email" variant="label-hidden"/>
            <lightning:outputField fieldName="Phone" variant="label-hidden"/>
        </lightning:recordViewForm>
    </div>
</lightning:card>
```

Additionally, to reduce the space between the label and field when the label variant is `label-inline`, use the `slds-form-element_1-col` class on `lightning:inputField`.

Usage Considerations

Admins can set the default display density for the org on the Density Settings setup page. Users can choose their own display density at any time. Admins can't override a user's display density setting. The org's default display setting depends on the Salesforce edition. Density changes don't apply to Salesforce Classic, Experience Builder sites, or the Salesforce mobile app. For more information, see Configure User Interface Settings.

SEE ALSO:

Component Library: `lightning:recordEditForm`

Component Library: `lightning:recordViewForm`

Considerations

Lightning Data Service is powerful and simple to use. However, it's not a complete replacement for writing your own data access code. Here are some considerations to keep in mind when using it.

Lightning Data Service is available in the following containers:

- Lightning Experience
- Salesforce app
- Experience Builder sites
- Lightning Out
- Lightning Components for Visualforce
- Standalone Lightning apps
- Lightning for Gmail
- Lightning for Outlook

Lightning Data Service supports primitive DML operations—create, read, update, and delete. It operates on one record at a time, which you retrieve or modify using the record ID. Lightning Data Service supports spanned fields with a maximum depth of five levels. Support for working with collections of records or for querying for a record by anything other than the record ID isn't available. If you must support higher-level operations or multiple operations in one transaction, use standard `@AuraEnabled` Apex methods.

Lightning Data Service shared data storage provides notifications to all components that use a record whenever a component changes that record. It doesn't notify components if that record is changed on the server, for example, if someone else modifies it. Records changed on the server aren't updated locally until they're reloaded. Lightning Data Service notifies listeners about data changes only if the changed fields are the same as in the listener's fields or layout.

Lightning Data Service does a lot of work to make code perform well.

- Loads record data progressively.
- Caches results on the client.
- Invalidates cache entries when dependent Salesforce data and metadata changes.
- Optimizes server calls by bulkifying and deduping requests.

Use Base Components

To work with record data, use the following base components.

- `lightning:recordForm`
- `lightning:recordEditForm`
- `lightning:recordViewForm`

Use these base components to:

- Create a metadata-driven UI or form-based UI similar to the record detail page in Salesforce.
- Display record values based on the field metadata.
- Display or hide localized field labels.
- Display the help text on a custom field.
- Perform client-side validation and enforce validation rules.

Alternatively, use `force:recordData` to:

- Create your own custom UI

- Return raw record data for a small number of fields
- Create UI that's not metadata-driven

When using `force:recordData`, load the data once and pass it to child components as attributes. This approach reduces the number of listeners and minimizes server calls, which improves performance and ensures that your components show consistent data. For more information, see the force:recordData documentation.

For examples of base components in action, see Lightning Action Examples on page 409.

The base components and `force:recordData` are built on Lightning Data Service. If Lightning Data Service detects a change to a record or any data or metadata it supports, the components receive the new value. The detection is triggered if:

- An Aura or Lightning web component mutates the record.
- The Lightning Data Service cache entry expires and then a component built on Lightning Data Service triggers a read. The cache entry and the Lightning web component must be in the same browser and app (for example Lightning Experience) for the same user.

Note: To improve performance, we recommend specifying the fields you need instead of using a layout. Use a layout only if you want the administrator, not the component, to control the fields that are provisioned. The component must handle receiving every field that is assigned to the layout for the context user. For more information, see Page Layouts in Salesforce Help.

Supported Objects

Lightning Data Service supports custom objects and the standard objects that User Interface API supports.

Lightning Action Examples

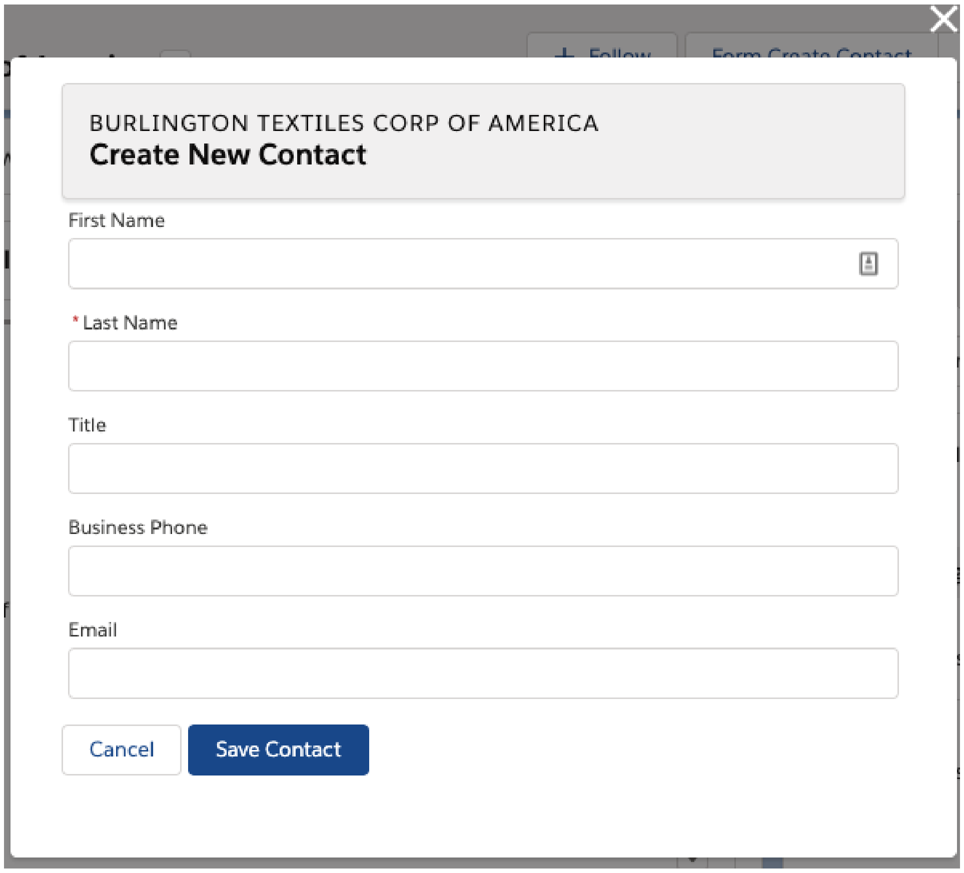
Here are some examples that use the base components to create a Quick Contact action panel.

Let's say you want to create a Lightning action that enables users to create contacts on an account record. You can do this easily using `lightning:recordViewForm` and `lightning:recordEditForm`. If you require granular customization, use `force:recordData`.

The following examples can each be added as a Lightning action on the account object. Clicking the action's button on the account layout opens a panel to create a contact.

Example: **Create a Lightning Action Using `lightning:recordViewForm` and `lightning:recordEditForm`**

The Quick Contact action panel includes a header with the account name and a form that creates a contact for that account record. Display the account name using `lightning:recordViewForm` and display the contact form using `lightning:recordEditForm`.


BURLINGTON TEXTILES CORP OF AMERICA
Create New Contact

First Name

* Last Name

Title

Business Phone

Email

formQuickContact.cmp

```
<aura:component implements="force:lightningQuickActionWithoutHeader,force:hasRecordId">


   <div class="slds-page-header" role="banner">
       <lightning:recordViewForm recordId="{!v.recordId}"
                                 objectApiName="Account">

           <div class="slds-text-heading_label">
               <lightning:outputField fieldName="Name" variant="label-hidden"/>
           </div>
           <lightning:messages/>
       </lightning:recordViewForm>
       <h1 class="slds-page-header__title slds-m-right_small
                      slds-truncate slds-align-left">Create New Contact</h1>
    </div>
    <lightning:recordEditForm aura:id="myform"
                              objectApiName="Contact"
                              onsubmit="{!c.handleSubmit}"
                              onsuccess="{!c.handleSuccess}">
        <lightning:messages/>
        <lightning:inputField fieldName="FirstName"/>
        <lightning:inputField fieldName="LastName"/>
        <lightning:inputField fieldName="Title"/>
        <lightning:inputField fieldName="Phone"/>
        <lightning:inputField fieldName="Email"/>
        <div class="slds-m-top_medium">
            <lightning:button label="Cancel" onclick="{!c.handleCancel}" />
            <lightning:button type="submit" label="Save Contact" variant="brand"/>
        </div>
    </lightning:recordEditForm>

</aura:component>
```

formQuickContactController.js

```
({
    handleCancel: function(cmp, event, helper) {
        $A.get("e.force:closeQuickAction").fire();
    },

    handleSubmit: function(cmp, event, helper) {
        event.preventDefault();
        var fields = event.getParam('fields');
        fields.AccountId = cmp.get("v.recordId");
        cmp.find('myform').submit(fields);
    },

    handleSuccess: function(cmp, event, helper) {
        // Success! Prepare a toast UI message
        var resultsToast = $A.get("e.force:showToast");
        resultsToast.setParams({
            "title": "Contact Saved",
            "message": "The new contact was created."
        });

        // Update the UI: close panel, show toast, refresh account page
        $A.get("e.force:closeQuickAction").fire();
        resultsToast.fire();

        // Reload the view
        $A.get("e.force:refreshView").fire();
    }
})
```

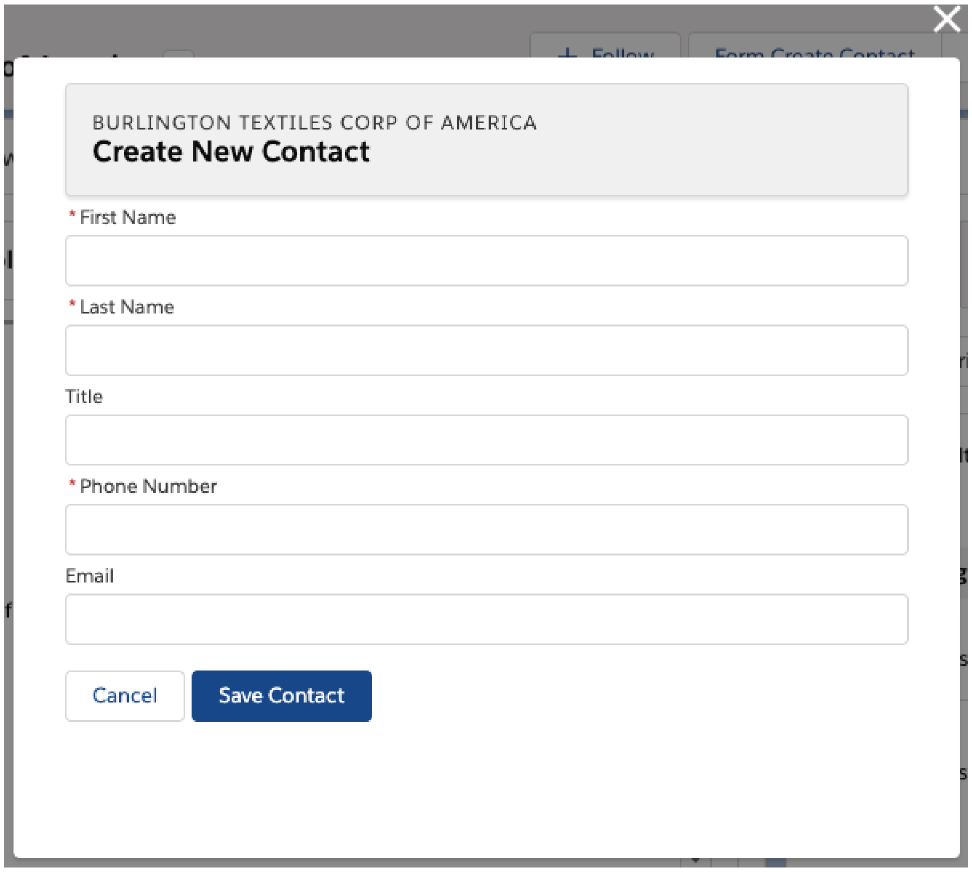
Using `lightning:recordEditForm`, you can nest the `lightning:inputField` components in `<div>` containers and add custom styling. You also need to provide your own cancel and submit buttons.

Consider the simpler `lightning:recordForm` component, which provides default **Cancel** and **Save** buttons. You can achieve the same result by replacing the `lightning:recordEditForm` component with the following.

```
<aura:attribute name="fields" type="String[]"
default="['FirstName','LastName','Title','Phone','Email']" />
<lightning:recordForm objectApiName="Contact"
                      fields="{!v.fields}"
                      onsubmit="{!c.handleSubmit}"
                      onsuccess="{!c.handleSuccess}" />
```

Example: **Create a Lightning Action Using `force:recordData`**

The Quick Contact action panel includes a header with the account name and a form that creates a contact for that account record. Display the account name and display the contact form using two separate instances of `force:recordData`.


BURLINGTON TEXTILES CORP OF AMERICA
Create New Contact
* First Name
* Last Name
Title
* Phone Number
Email

This `force:recordData` example is similar to the example provided in Configure Components for Record-Specific Actions. Compare the two examples to better understand the differences between using `@AuraEnabled` Apex controllers and using Lightning Data Service.

`ldsQuickContact.cmp`

```
<aura:component implements="force:lightningQuickActionWithoutHeader,force:hasRecordId">


    <aura:attribute name="account" type="Object"/>
    <aura:attribute name="simpleAccount" type="Object"/>
    <aura:attribute name="accountError" type="String"/>
    <force:recordData aura:id="accountRecordLoader"
        recordId="{!v.recordId}"
        fields="Name,BillingCity,BillingState"
        targetRecord="{!v.account}"
        targetFields="{!v.simpleAccount}"
        targetError="{!v.accountError}"
    />
    <aura:attribute name="newContact" type="Object" access="private"/>
    <aura:attribute name="simpleNewContact" type="Object" access="private"/>
    <aura:attribute name="newContactError" type="String" access="private"/>
    <force:recordData aura:id="contactRecordCreator"
        layoutType="FULL"
        targetRecord="{!v.newContact}"
        targetFields="{!v.simpleNewContact}"
        targetError="{!v.newContactError}"
        />

    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <!-- Display a header with details about the account -->
    <div class="slds-page-header" role="banner">
        <p class="slds-text-heading_label">{!v.simpleAccount.Name}</p>
        <h1 class="slds-page-header__title slds-m-right_small
            slds-truncate slds-align-left">Create New Contact</h1>
    </div>

    <!-- Display Lightning Data Service errors, if any -->
    <aura:if isTrue="{!not(empty(v.accountError))}">
            {!v.accountError}
    </aura:if>
    <aura:if isTrue="{!not(empty(v.newContactError))}">
        {!v.newContactError}
    </aura:if>

    <!-- Display the new contact form -->
    <lightning:input aura:id="contactField" name="firstName" label="First Name"
                     value="{!v.simpleNewContact.FirstName}" required="true"/>

    <lightning:input aura:id="contactField" name="lastname" label="Last Name"
                     value="{!v.simpleNewContact.LastName}" required="true"/>

    <lightning:input aura:id="contactField" name="title" label="Title"
                     value="{!v.simpleNewContact.Title}" />

    <lightning:input aura:id="contactField" type="phone" name="phone" label="Phone
Number"
                     pattern="^(1?(-?\d{3})-?)?(\d{3})(-?\d{4})$"
                     messageWhenPatternMismatch="The phone number must contain 7, 10,
 or 11 digits. Hyphens are optional."
                     value="{!v.simpleNewContact.Phone}" required="true"/>

    <lightning:input aura:id="contactField" type="email" name="email" label="Email"
                     value="{!v.simpleNewContact.Email}" />

    <lightning:button label="Cancel" onclick="{!c.handleCancel}"
class="slds-m-top_medium" />
    <lightning:button label="Save Contact" onclick="{!c.handleSaveContact}"
                      variant="brand" class="slds-m-top_medium"/>


</aura:component>
```

ldsQuickContactController.js

```
({
    doInit: function(component, event, helper) {
        component.find("contactRecordCreator").getNewRecord(
            "Contact", // objectApiName
            null, // recordTypeId
            false, // skip cache?
            $A.getCallback(function() {
                var rec = component.get("v.newContact");
                var error = component.get("v.newContactError");
                if(error || (rec === null)) {
                    console.log("Error initializing record template: " + error);
                }
                else {
                    console.log("Record template initialized: " + rec.apiName);
                }
            })
        );
    },

    handleSaveContact: function(component, event, helper) {
        if(helper.validateContactForm(component)) {
          component.set("v.simpleNewContact.AccountId", component.get("v.recordId"));

            component.find("contactRecordCreator").saveRecord(function(saveResult) {
                if (saveResult.state === "SUCCESS" || saveResult.state === "DRAFT") {


                    // Success! Prepare a toast UI message
                    var resultsToast = $A.get("e.force:showToast");
                    resultsToast.setParams({
                        "title": "Contact Saved",
                        "message": "The new contact was created."
                    });

                    // Update the UI: close panel, show toast, refresh account page
                    $A.get("e.force:closeQuickAction").fire();
                    resultsToast.fire();

                    // Reload the view so components not using force:recordData
                    // are updated
                    $A.get("e.force:refreshView").fire();
                }
                else if (saveResult.state === "INCOMPLETE") {
                    console.log("User is offline, device doesn't support drafts.");
                }
                else if (saveResult.state === "ERROR") {
                    console.log('Problem saving contact, error: ' +
                                 JSON.stringify(saveResult.error));
                }
                else {
                    console.log('Unknown problem, state: ' + saveResult.state +
                                ', error: ' + JSON.stringify(saveResult.error));
                }
            });
        }
    },

    handleCancel: function(component, event, helper) {
        $A.get("e.force:closeQuickAction").fire();
    },
})
```

Note: The callback passed to `getNewRecord()` must be wrapped in `$A.getCallback()` to ensure correct access context when the callback is invoked. If the callback is passed in without being wrapped in `$A.getCallback()`, any attempt to access private attributes of your component results in access check failures.

Even if you're not accessing private attributes, it's a best practice to always wrap the callback function for `getNewRecord()` in `$A.getCallback()`. Never mix (contexts), never worry.

`ldsQuickContactHelper.js`

```
({
    validateContactForm: function(component) {
        var validContact = true;

         // Show error messages if required fields are blank
        var allValid = component.find('contactField').reduce(function (validFields,
inputCmp) {
            inputCmp.showHelpMessageIfInvalid();
            return validFields && inputCmp.get('v.validity').valid;
        }, true);

        if (allValid) {
            // Verify we have an account to attach it to
            var account = component.get("v.account");
            if($A.util.isEmpty(account)) {
                validContact = false;
                console.log("Quick action context doesn't have a valid account.");
            }
            return(validContact);
        }
    }
})
```

Usage Differences

Consider the following differences between the previous examples.

Field labels and values

`lightning:recordViewForm` and `lightning:recordEditForm` obtain labels and the requiredness properties from the object schema. In the first example, the `Last Name` field is a required field on the contact object. The component provides field-level validation.

With `force:recordData`, you must provide your own labels and requiredness property for each field. You can also provide your own field-level validation, as shown by the `lightning:input` component with the `pattern` and `messageWhenPatternMismatch` attributes.

Saving the record

`lightning:recordEditForm` saves the record automatically when you provide a `lightning:button` component with the `submit` type.

With `force:recordData`, you must call the `saveRecord` function.

Lightning Data Service errors

`lightning:recordViewForm` and `lightning:recordEditForm` display Lightning Data Service errors automatically using `lightning:messages`, and provide custom error handling via the `onerror` event handler.

With `force:recordData`, you must handle and display the errors on your own.

SEE ALSO:

Configure Components for Record-Specific Actions

Controlling Access

`SaveRecordResult`

Represents the result of a Lightning Data Service operation that makes a persistent change to record data.

`SaveRecordResult` Object

Callback functions for the `saveRecord` and `deleteRecord` functions receive a `SaveRecordResult` object as their only argument.

Attribute Name	Type	Description
`objectApiName`	String	The object API name for the record.
`entityLabel`	String	The label for the name of the sObject of the record.
`error`	String	Error is one of the following. • A localized message indicating what went wrong. • An array of errors, including a localized message indicating what went wrong. It might also include further data to help handle the error, such as field- or page-level errors. `error` is undefined if the save `state` is SUCCESS or DRAFT.
`recordId`	String	The 18-character ID of the record affected.
`state`	String	The result state of the operation. Possible values are: • SUCCESS—The operation completed on the server successfully. • DRAFT—The server wasn't reachable, so the operation was saved locally as a draft. The change is applied to the server when it's reachable. • INCOMPLETE—The server wasn't reachable, and the device doesn't support drafts. (Drafts are supported only in the Salesforce app.) Try this operation again later. • ERROR—The operation couldn't be completed. Check the `error` attribute for more information.

Displaying the Create and Edit Record Modals

You can take advantage of built-in events to display modals that let you create or edit records via an Aura component.

The `force:createRecord` and `force:editRecord` events display a create record page and edit record page in a modal based on the default custom layout type for that object.

The following example contains a button that calls a client-side controller to display the edit record page. Add this example component to a record page to inherit the record ID via the `force:hasRecordId` interface.

```
<aura:component implements="flexipage:availableForRecordHome,force:hasRecordId" >
    <aura:attribute name="recordId" type="String" />
    <lightning:button label="Edit Record" onclick="{!c.edit}"/>
</aura:component>
```

The client-side controller fires the `force:editRecord` event, which displays the edit record page for a given record ID.

```
edit : function(component, event, helper) {
    var editRecordEvent = $A.get("e.force:editRecord");
    editRecordEvent.setParams({
        "recordId": component.get("v.recordId")
    });
    editRecordEvent.fire();
}
```

Firing this event on a record page is similar to clicking the default Edit button on a record page's header. Records updated using the `force:editRecord` event are persisted automatically.

Note: If you don't need the edit record page to display in a modal or if you need to specify a subset of fields, consider using Lightning Data Service via `lightning:recordForm` or `lightning:recordEditForm` instead.

Using Apex

Use Apex to write server-side code, such as controllers and test classes. Use Apex only if you need to customize your user interface to do more than what Lightning Data Service allows, such as using a SOQL query to select certain records. Apex provisions data that's not managed and you must handle data refresh on your own.

Apex controllers handle requests from client-side controllers. For example, a client-side controller might handle an event and call an Apex controller action to persist a record. An Apex controller can also load your record data.

Use Apex in these scenarios:

- To work with objects that aren't supported by User Interface API, such as Task and Event.
- To work with operations that User Interface API doesn't support, like loading a list of records by criteria (for example, to load the first 200 Accounts with Amount > $1M).
- To perform a transactional operation. For example, to create an account and create an opportunity associated with the new account. If either create fails, the entire transaction is rolled back.
- To call a method imperatively, such as in response to clicking a button, or to delay loading to outside the critical path.

IN THIS SECTION:

Creating Server-Side Logic with Controllers

The framework supports client-side (JavaScript) and server-side (Apex) controllers. An event is always wired to a client-side controller action, which can in turn call an Apex controller action. For example, a client-side controller might handle an event and call an Apex controller action to persist a record.

Testing Your Apex Code

Before you can upload a managed package, you must write and execute tests for your Apex code to meet minimum code coverage requirements. Also, all tests must run without errors when you upload your package to AppExchange.

Making API Calls from Apex

Make API calls from an Apex controller. You can't make Salesforce API calls from JavaScript code.

Make Long-Running Callouts with Continuations

Use the `Continuation` class in Apex to make a long-running request to an external web service. Process the response in a callback method. Continuations are the preferred way to manage callouts because they can provide substantial improvements to the user experience.

Creating Components in Apex

Creating components on the server side in Apex, using the `Cmp.<myNamespace>.<myComponent>` syntax, is deprecated. Use `$A.createComponent()` in client-side JavaScript code instead.

Creating Server-Side Logic with Controllers

The framework supports client-side (JavaScript) and server-side (Apex) controllers. An event is always wired to a client-side controller action, which can in turn call an Apex controller action. For example, a client-side controller might handle an event and call an Apex controller action to persist a record.

Server-side actions need to make a round trip, from the client to the server and back again, so they usually complete more slowly than client-side actions.

For more details on the process of calling a server-side action, see Calling a Server-Side Action on page 428.

IN THIS SECTION:

Apex Server-Side Controller Overview

Create a server-side controller in Apex and use the `@AuraEnabled` annotation to enable access to the controller method.

AuraEnabled Annotation

The `AuraEnabled` annotation enables Lightning components to access Apex methods and properties.

Creating an Apex Server-Side Controller

Use the Developer Console to create an Apex server-side controller.

Using Apex to Work with Salesforce Records

Use Apex only if you need to customize your user interface to do more than what Lightning Data Service allows, such as using a SOQL query to select certain records. Apex provisions data that's not managed and you must handle data refresh on your own.

Granting User Access for Apex Classes

An authenticated or guest user can access an `@AuraEnabled` Apex method only when the user's profile or an assigned permission set allows access to the Apex class.

Securing Data in Apex Controllers

By default, Apex runs in system mode, which means that it runs with substantially elevated permissions, acting as if the user had most permissions and all field- and object-level access granted. Because these security layers aren't enforced like they are in the Salesforce UI, you must write code to enforce them. Otherwise, your components may inadvertently expose sensitive data that would normally be hidden from users in the Salesforce UI.

Calling a Server-Side Action

Call a server-side controller action from a client-side controller. In the client-side controller, you set a callback, which is called after the server-side action is completed. A server-side action can return any object containing serializable JSON data.

Passing Data to an Apex Controller

Use `action.setParams()` in JavaScript to set data to pass to an Apex controller.

Returning Data from an Apex Server-Side Controller

Return results from a server-side controller to a client-side controller using the `return` statement. Results data must be serializable into JSON format.

Returning Errors from an Apex Server-Side Controller

Create and throw a `System.AuraHandledException` from your Apex controller to return a custom error message to a JavaScript controller.

Queueing of Server-Side Actions

The framework queues up actions before sending them to the server. This mechanism is largely transparent to you when you're writing code but it enables the framework to minimize network traffic by batching multiple actions into one request (XHR).

Foreground and Background Actions

Foreground actions are the default. An action can be marked as a background action. This is useful when you want your app to remain responsive to a user while it executes a low priority, long-running action. A rough guideline is to use a background action if it takes more than five seconds for the response to return from the server.

Storable Actions

Enhance your component's performance by marking actions as storable (cacheable) to quickly show cached data from client-side storage without waiting for a server trip. If the cached data is stale, the framework retrieves the latest data from the server. Caching is especially beneficial for users on high latency, slow, or unreliable connections such as 3G networks.

Abortable Actions

Mark an action as abortable to make it potentially abortable while it's queued to be sent to the server. An abortable action in the queue is not sent to the server if the component that created the action is no longer valid, that is `cmp.isValid() == false`. A component is automatically destroyed and marked invalid by the framework when it is unrendered.

Apex Server-Side Controller Overview

Create a server-side controller in Apex and use the `@AuraEnabled` annotation to enable access to the controller method.

Only methods that you have explicitly annotated with `@AuraEnabled` are exposed. Calling server-side actions aren't counted against your org's API limits. However, your server-side controller actions are written in Apex, and as such are subject to all the usual Apex limits. Apex limits are applied per action.

This Apex controller contains a `serverEcho` action that prepends a string to the value passed in.

```
public with sharing class SimpleServerSideController {

    //Use @AuraEnabled to enable client- and server-side access to the method
    @AuraEnabled
    public static String serverEcho(String firstName) {
        return ('Hello from the server, ' + firstName);
    }
}
```

In addition to using the `@AuraEnabled` annotation, your Apex controller must follow these requirements.

- Methods must be `static` and marked `public` or `global`. Non-static methods aren't supported.
- If a method returns an object, instance methods that retrieve the value of the object's instance field must be `public`.
- Use unique names for client-side and server-side actions in a component. A JavaScript function (client-side action) with the same name as an Apex method (server-side action) can lead to hard-to-debug issues. In debug mode, the framework logs a browser console warning about the clashing client-side and server-side action names.

Tip: Don't store component state in your controller (client-side or server-side). Store state in a component's client-side attributes instead.

For more information, see Classes in the *Apex Developer Guide*.

SEE ALSO:

Calling a Server-Side Action

Creating an Apex Server-Side Controller

AuraEnabled Annotation

Apex Class Considerations for Packages

`AuraEnabled` Annotation

The `AuraEnabled` annotation enables Lightning components to access Apex methods and properties.

The `AuraEnabled` annotation is overloaded, and is used for two separate and distinct purposes.

- Use `@AuraEnabled` on Apex **class static methods** to make them accessible as remote controller actions in your Lightning components.
- Use `@AuraEnabled` on Apex **instance methods and properties** to make them serializable when an instance of the class is returned as data from a server-side action.

Important:

- Don't mix-and-match these different uses of `@AuraEnabled` in the same Apex class.
- Only static `@AuraEnabled` Apex methods can be called from client-side code. Visualforce-style instance properties and getter/setter methods aren't available. Use client-side component attributes instead.
- You can't use an Apex inner class as a parameter or return value for an Apex method that's called by an Aura component.
- You can't use the `@NamespaceAccessible` Apex annotation for an `@AuraEnabled` Apex method referenced from an Aura component.

Component Security

In Apex, every method that is annotated `@AuraEnabled` should be treated as a web service interface. That is, the developer should assume that an attacker can call this method with any parameter, even if the developer's client-side code does not invoke the method or invokes it using only sanitized parameters. For more information, see the Secure Coding Guide.

For API version of 50.0 or higher, you must specify which users can access Apex classes that contain `@AuraEnabled` methods. For more information, see Salesforce Developers Blog: Breezing Through the Upcoming @AuraEnabled Critical Update.

Caching Method Results

To improve runtime performance, set `@AuraEnabled(cacheable=true)` to cache the method results on the client. To set `cacheable=true`, a method must only get data. It can't mutate data.

Marking a method as storable (cacheable) improves your component's performance by quickly showing cached data from client-side storage without waiting for a server trip. If the cached data is stale, the framework retrieves the latest data from the server. Caching is especially beneficial for users on high latency, slow, or unreliable connections such as 3G networks.

To cache data returned from an Apex method for any component with an API version of 44.0 or higher, you must annotate the Apex method with `@AuraEnabled(cacheable=true)`. For example:

```
@AuraEnabled(cacheable=true)
public static Account getAccount(Id accountId) {
    // your code here
}
```

Prior to API version 44.0, to cache data returned from an Apex method, you had to call `setStorable()` in JavaScript code on every action that called the Apex method. For API version of 44.0 or higher, you must mark the Apex method as storable (cacheable) and you can get rid of any `setStorable()` calls in JavaScript code. The Apex annotation approach is better because it centralizes your caching notation for a method in the Apex class.

Note: Client-side storage is automatically configured in Lightning Experience and the Salesforce mobile app. A component shouldn't assume a cache duration because it may change as we optimize the platform.

Example: The `AccountController.cls` Apex class from the github.com/trailheadapps/lwc-recipes repo shows how to use `@AuraEnabled(cacheable=true)`.

Using Continuations

Use the `Continuation` class in Apex to make a long-running request to an external Web service.

Continuations use the `@AuraEnabled` annotation. Here are the rules for usage.

`@AuraEnabled(continuation=true)`
An Apex controller method that returns a continuation must be annotated with `@AuraEnabled(continuation=true)`.

`@AuraEnabled(continuation=true cacheable=true)`
To cache the result of a continuation action, set `cacheable=true` on the annotation for the Apex callback method.

Note: There's a space, **not a comma**, between `continuation=true cacheable=true`.

SEE ALSO:

Returning Data from an Apex Server-Side Controller
Custom Apex Class Types
Storable Actions
Securing Data in Apex Controllers
@AuraEnabled Annotations for Continuations
Apex Developer Guide: NamespaceAccessible Annotation

Creating an Apex Server-Side Controller

Use the Developer Console to create an Apex server-side controller.

1. Open the Developer Console.
2. Click **File** > **New** > **Apex Class**.
3. Enter a name for your server-side controller.
4. Click **OK**.
5. Enter a method for each server-side action in the body of the class.

   Add the `@AuraEnabled` annotation to a method to expose it as a server-side action. Additionally, server-side actions must be `static` methods, and either `global` or `public`.
6. Click **File** > **Save**.
7. Open the component that you want to wire to the new controller class.
8. Add a `controller` system attribute to the `<aura:component>` tag to wire the component to the controller. For example:

   ```
   <aura:component controller="SimpleServerSideController">
   ```

SEE ALSO:

Salesforce Help: Open the Developer Console

Returning Data from an Apex Server-Side Controller

AuraEnabled Annotation

Granting User Access for Apex Classes

Apex Class Considerations for Packages

Using Apex to Work with Salesforce Records

Use Apex only if you need to customize your user interface to do more than what Lightning Data Service allows, such as using a SOQL query to select certain records. Apex provisions data that's not managed and you must handle data refresh on your own.

The term `sObject` refers to any object that can be stored in Lightning Platform. This could be a standard object, such as Account, or a custom object that you create, such as a Merchandise object.

An `sObject` variable represents a row of data, also known as a record. To work with an object in Apex, declare it using the SOAP API name of the object. For example:

```
Account a = new Account();
MyCustomObject__c co = new MyCustomObject__c();
```

For more information on working on records with Apex, see Working with Data in Apex.

This example controller persists an updated Account record. Note that the `update` method has the `@AuraEnabled` annotation, which enables it to be called as a server-side controller action.

```
public with sharing class AccountController {

    @AuraEnabled
    public static void updateAnnualRevenue(String accountId, Decimal annualRevenue) {
        Account acct = [SELECT Id, Name, BillingCity FROM Account WHERE Id = :accountId];

        acct.AnnualRevenue = annualRevenue;

        // Perform isAccessible() and isUpdateable() checks here
        update acct;
```

```
    }
}
```

Note: When using Apex controllers, load the data once and pass it to child components as attributes. This approach reduces the number of listeners and minimizes server calls, which improves performance and ensures that your components show consistent data.

Differences Between Lightning Data Service and Apex

The `lightning:record*Form` on page 384 and `force:recordData` components are the easiest way to work with records. They are built on top of Lightning Data Service, which manages field-level security and sharing for you in addition to managing data loading and refresh. You can use these components for objects that are supported by User Interface API

Use Apex only if you're working with a scenario listed at Using Apex on page 417, You can call the Apex method imperatively, such as in response to a button click, as shown in the **Loading Record Data from a Standard Object** section. Alternatively, to load record data during component initialization, use the `init` handler, as shown in the **Loading Record Data By Criteria** section. When using Apex to load or provision data, you must handle data refresh on your own by invoking the Apex method again.

Loading Record Data from an Object

Load records from an object in an Apex controller. The following Apex controller has methods that return a list of tasks. Task is an object that isn't supported by Lightning Data Service and the User Interface API. Therefore, we recommend using Apex to load task record data.

```
public with sharing class TaskController {

    @AuraEnabled(cacheable=true)
    public static List<Task> getTasks() {
        return [SELECT Subject, Priority, Status FROM Task];    }
}
```

This example component uses the previous Apex controller to display a list of task record data when you press a button. The `flexipage:availableForAllPageTypes` interface denotes that you can use this example on a Lightning page.

```
<!-- apexForTasks.cmp -->
<aura:component implements="flexipage:availableForAllPageTypes" controller="TaskController">

    <aura:attribute name="tasks" type="Task[]"/>
    <lightning:card iconName="standard:task">

        <lightning:button label="Get Tasks" onclick="{!c.getMyTasks}"/>
        <aura:iteration var="task" items="{!v.tasks}">
            <p>{!task.Subject} : {!task.Priority}, {!task.Status}</p>
        </aura:iteration>

    </lightning:card>
</aura:component>
```

When you press the button, the following client-side controller calls the `getTasks()` method and sets the `tasks` attribute on the component. For more information about calling server-side controller methods, see Calling a Server-Side Action on page 428.

```
// apexForTasksController.js
({
    getMyTasks: function(cmp){
        var action = cmp.get("c.getTasks");
        action.setCallback(this, function(response){
            var state = response.getState();
            if (state === "SUCCESS") {
                cmp.set("v.tasks", response.getReturnValue());
            }
        });
  $A.enqueueAction(action);
    }
})
```

Loading Record Data By Criteria

As we've learned, to load a simple list of record data, you can use base components or `force:recordData`, as shown at Loading a Record on page 385. But to use a SOQL query to select certain records, use an Apex controller.

Remember that the method must be `static`, and `global` or `public`. The method must be decorated with `@AuraEnabled(cacheable=true)`.

For example, query related cases based on an account Id and limit the result to 10 records.

```
public with sharing class CaseController {
    @AuraEnabled(cacheable=true)
    public static List<Case> getCases(String accountId) {
        return [SELECT AccountId, Id, Subject, Status, Priority, CaseNumber
                FROM Case
                WHERE AccountId = :accountId LIMIT 10];
    }
}
```

The client-side controller loads related cases using the `init` handler. The `action.setParams()` method passes in the record Id of the account record being viewed to the Apex controller,

```
// casesForAccountController.js
({
    init : function(cmp, evt) {
        var action = cmp.get("c.getCases");
        action.setParams({
            "accountId": cmp.get("v.recordId")
        });
        action.setCallback(this, function(response){
            var state = response.getState();
            if (state === "SUCCESS") {
                cmp.set("v.cases", response.getReturnValue());
            }
        });
        $A.enqueueAction(action);
    }
})
```

In your custom component, load a form that enables editing and updating of cases on an account record using `lightning:recordEditForm`, by performing these steps.

- Query the relevant cases and set the result to the component attribute `v.cases`.
- Iterate over the cases by passing in the case Id to the `recordId` attribute on `lightning:recordEditForm`.

The example implements the `flexipage:availableForRecordHome` and `force:hasRecordId` interfaces so you can use the example on an account record page.

```
<!-- casesForAccount.cmp -->
<aura:component implements="flexipage:availableForRecordHome,force:hasRecordId"
controller="CaseController">
    <aura:attribute name="cases" type="Case[]"/>
    <aura:attribute name="recordId" type="Id" />
    <aura:handler name="init" value="{! this }" action="{! c.init }"/>

    <aura:iteration items="{!v.cases}" var="case">
        <lightning:card title="{!case.Id}" iconName="standard:case">
            <lightning:recordEditForm objectApiName="Case" recordId="{!case.Id}">
                <lightning:inputField fieldName="Subject"/>
                <lightning:inputField fieldName="Status"/>

                <!- Read-only field -->
                <lightning:outputField fieldName="Origin" variant="label-hidden"/>

                <lightning:button label="Update case" type="submit"/>
            </lightning:recordEditForm>
        </lightning:card>
    </aura:iteration>
</aura:component>
```

Note: The case data on the account record is managed by Lightning Data Service since it uses `lightning:recordEditForm`; therefore, the case data that's referenced (subject, status, and origin) reflects the latest data. However, if a case on the account is deleted or a new case is added to the account, you must invoke the Apex method again to query the new results.

For read-only data, use `lightning:outputField`. To work with read-only data only, use `lightning:recordViewForm` or `lightning:recordForm`. For granular control of your UI, use `force:recordData`. For more information, see Lightning Data Service on page 384.

SEE ALSO:

Securing Data in Apex Controllers

Granting User Access for Apex Classes

An authenticated or guest user can access an `@AuraEnabled` Apex method only when the user's profile or an assigned permission set allows access to the Apex class.

For details on configuring user profile or permission set access to an Apex class, see Class Security in the Apex Developer Guide.

SEE ALSO:

Creating an Apex Server-Side Controller

AuraEnabled Annotation

Securing Data in Apex Controllers

Securing Data in Apex Controllers

By default, Apex runs in system mode, which means that it runs with substantially elevated permissions, acting as if the user had most permissions and all field- and object-level access granted. Because these security layers aren't enforced like they are in the Salesforce UI, you must write code to enforce them. Otherwise, your components may inadvertently expose sensitive data that would normally be hidden from users in the Salesforce UI.

Note: To work with Salesforce records, we recommend using Lightning Data Service, which handles sharing rules, CRUD, and field-level security for you.

Enforce Sharing Rules

When you declare a class, it's a best practice to specify `with sharing` to enforce sharing rules when a component uses the Apex controller.

```
public with sharing class SharingClass {
    // Code here
}
```

An `@AuraEnabled` Apex class that doesn't explicitly set `with sharing` or `without sharing`, or is defined with `inherited sharing`, uses a default or implicit value of `with sharing`. However, an Apex class that doesn't explicitly set `with sharing` or `without sharing` inherits the value from the context in which it runs. So when a class without explicit sharing behavior is called by a class that sets one of the keywords, it operates with the sharing behavior of the calling class. To ensure that your class enforces sharing rules, set `with sharing`.

The `with sharing` keyword enforces record-level security. It doesn't enforce object-level and field-level security. You must manually enforce object-level and field-level security separately in your Apex classes.

Enforce Object and Field Permissions (CRUD and FLS)

There are a few alternatives to enforce object-level and field-level permissions in your Apex code.

Easiest enforcement using `WITH USER_MODE`

To enforce object-level and field-level permissions, use the `WITH USER_MODE` clause for `SOQL SELECT` queries in Apex code, including subqueries and cross-object relationships.

The `WITH USER_MODE` clause is ideal if you have minimal experience developing secure code and for applications that don't require graceful degradation on permissions errors.

This example queries fields on a custom expense object with an insecure method, `get_UNSAFE_Expenses()`. Don't use this class!

```
// This class is an anti-pattern.
public with sharing class UnsafeExpenseController {
    // ns refers to namespace; leave out ns__ if not needed
    // This method is vulnerable because it doesn't enforce FLS.
    @AuraEnabled
    public static List<ns__Expense__c> get_UNSAFE_Expenses() {
        return [SELECT Id, Name, ns__Amount__c, ns__Client__c, ns__Date__c,
            ns__Reimbursed__c, CreatedDate FROM ns__Expense__c];
    }
}
```

This next example uses a secure method, `getExpenses()`, which uses the `WITH USER_MODE` clause to enforce object-level and field-level permissions. Use this class instead of `UnsafeExpenseController`.

```
public with sharing class ExpenseController {
    // This method is recommended because it enforces FLS.
    @AuraEnabled
    public static List<ns__Expense__c> getExpenses() {
    // Query the object safely
    return [SELECT Id, Name, ns__Amount__c, ns__Client__c, ns__Date__c,
            ns__Reimbursed__c, CreatedDate
             FROM ns__Expense__c WITH USER_MODE];
    }
}
```

For more details, see Enforce User Mode for Database Operations in the *Apex Developer Guide*.

Graceful degradation with `stripInaccessible()`

For more graceful degradation on permissions errors, use the `stripInaccessible()` method to enforce field- and object-level data protection. This method strips the fields and relationship fields from query and subquery results that the user can't access. You can find out if any fields were stripped and throw an `AuraHandledException` with a custom error message, if desired.

You can also use the method to remove inaccessible sObject fields before DML operations to avoid exceptions and to sanitize sObjects that have been deserialized from an untrusted source.

This example updates `ExpenseController` to use `stripInaccessible()` instead of the `WITH USER_MODE` SOQL clause. The results are the same but `stripInaccessible()` gives you the opportunity to gracefully degrade instead of failing on an access violation when using `WITH USER_MODE`.

```
public with sharing class ExpenseControllerStripped {

    @AuraEnabled
    public static List<ns__Expense__c> getExpenses() {
        // Query the object but don't use WITH USER_MODE
        List<ns__Expense__c> expenses =
            [SELECT Id, Name, ns__Amount__c, ns__Client__c, ns__Date__c,
                ns__Reimbursed__c, CreatedDate
                 FROM ns__Expense__c];

         // Strip fields that are not readable
         SObjectAccessDecision decision = Security.stripInaccessible(
               AccessType.READABLE,
               expenses);

         // Throw an exception if any data was stripped
         if (!decision.getModifiedIndexes().isEmpty()) {
            throw new AuraHandledException('Data was stripped');
         }

         return expenses;
    }
}
```

For more details and examples, see Enforce Security with the stripInaccessible Method in the *Apex Developer Guide*.

Legacy code using `DescribeSObjectResult` and `DescribeFieldResult` methods

Before the `WITH USER_MODE` clause and `stripInaccessible()` method were available, the only way to enforce object and field permissions was to check the current user's access permission levels by calling the `Schema.DescribeSObjectResult` and `Schema.DescribeFieldResult` methods. Then, if a user has the necessary permissions, perform a specific DML operation or a query.

For example, you can call the `isAccessible`, `isCreateable`, or `isUpdateable` methods of `Schema.DescribeSObjectResult` to verify whether the current user has read, create, or update access to an `sObject`, respectively. Similarly, `Schema.DescribeFieldResult` exposes access control methods that you can call to check the current user's read, create, or update access for a field.

This example uses the describe result methods. This approach requires many more lines of boilerplate code so we recommend using the `WITH USER_MODE` clause or `stripInaccessible()` method instead.

```
public with sharing class ExpenseControllerLegacy {
    @AuraEnabled
    public static List<ns__Expense__c> getExpenses() {
        String [] expenseAccessFields = new String [] {'Id',
                                                       'Name',
                                                       'ns__Amount__c',
                                                       'ns__Client__c',
                                                       'ns__Date__c',
                                                       'ns__Reimbursed__c',
                                                       'CreatedDate'
                                                       };


    // Obtain the field name/token map for the Expense object
   Map<String,Schema.SObjectField> m = Schema.SObjectType.ns__Expense__c.fields.getMap();


    for (String fieldToCheck : expenseAccessFields) {

        // Call getDescribe to check if the user has access to view field
        if (!m.get(fieldToCheck).getDescribe().isAccessible()) {

            // Pass error to client
            throw new System.NoAccessException();
        }
    }

    // Query the object safely
    return [SELECT Id, Name, ns__Amount__c, ns__Client__c, ns__Date__c,
            ns__Reimbursed__c, CreatedDate FROM ns__Expense__c];
    }
}
```

SEE ALSO:

Apex Developer Guide: Enforcing Sharing Rules

Apex Developer Guide: Enforcing Object and Field Permissions

Apex Developer Guide: Using the with sharing, without sharing, and inherited sharing Keywords

Calling a Server-Side Action

Call a server-side controller action from a client-side controller. In the client-side controller, you set a callback, which is called after the server-side action is completed. A server-side action can return any object containing serializable JSON data.

A client-side controller is a JavaScript object in object-literal notation containing a map of name-value pairs.

Let's say that you want to trigger a server-call from a component. The following component contains a button that's wired to a client-side controller `echo` action. `SimpleServerSideController` contains a method that returns a string passed in from the client-side controller.

```
<aura:component controller="SimpleServerSideController">
    <aura:attribute name="firstName" type="String" default="world"/>
    <lightning:button label="Call server" onclick="{!c.echo}"/>
</aura:component>
```

This client-side controller includes an `echo` action that executes a `serverEcho` method on a server-side controller.

Tip: Use unique names for client-side and server-side actions in a component. A JavaScript function (client-side action) with the same name as an Apex method (server-side action) can lead to hard-to-debug issues. In debug mode, the framework logs a browser console warning about the clashing client-side and server-side action names.

```
({
    "echo" : function(cmp) {
        // create a one-time use instance of the serverEcho action
        // in the server-side controller
        var action = cmp.get("c.serverEcho");
        action.setParams({ firstName : cmp.get("v.firstName") });

        // Create a callback that is executed after
        // the server-side action returns
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                // Alert the user with the value returned
                // from the server
                alert("From server: " + response.getReturnValue());

                // You would typically fire a event here to trigger
                // client-side notification that the server-side
                // action is complete
            }
            else if (state === "INCOMPLETE") {
                // do something
            }
            else if (state === "ERROR") {
                var errors = response.getError();
                if (errors) {
                    if (errors[0] && errors[0].message) {
                        console.log("Error message: " +
                                 errors[0].message);
                    }
                } else {
                    console.log("Unknown error");
                }
            }
        });

        // optionally set storable, abortable, background flag here

        // A client-side action could cause multiple events,
        // which could trigger other events and
        // other server-side action calls.
        // $A.enqueueAction adds the server-side action to the queue.
        $A.enqueueAction(action);
    }
})
```

In the client-side controller, we use the value provider of `c` to invoke a server-side controller action. We also use the `c` syntax in markup to invoke a client-side controller action.

The `cmp.get("c.serverEcho")` call indicates that we're calling the `serverEcho` method in the server-side controller. The method name in the server-side controller must match everything after the `c.` in the client-side call. In this case, that's `serverEcho`.

The implementation of the `serverEcho` Apex method is shown in Apex Server-Side Controller Overview.

Use `action.setParams()` to set data to be passed to the server-side controller. The following call sets the value of the `firstName` argument on the server-side controller's `serverEcho` method based on the `firstName` attribute value.

```
action.setParams({ firstName : cmp.get("v.firstName") });
```

`action.setCallback()` sets a callback action that is invoked after the server-side action returns.

```
action.setCallback(this, function(response) { ... });
```

The server-side action results are available in the `response` variable, which is the argument of the callback.

`response.getState()` gets the state of the action returned from the server.

Note: You don't need a `cmp.isValid()` check in the callback in a client-side controller when you reference the component associated with the client-side controller. The framework automatically checks that the component is valid.

`response.getReturnValue()` gets the value returned from the server. In this example, the callback function alerts the user with the value returned from the server.

`$A.enqueueAction(action)` adds the server-side controller action to the queue of actions to be executed. All actions that are enqueued will run at the end of the event loop. Rather than sending a separate request for each individual action, the framework processes the event chain and batches the actions in the queue into one request. The actions are asynchronous and have callbacks.

Tip: If your action isn't executing, make sure that you're not executing code outside the framework's normal rendering lifecycle. For example, if you use `window.setTimeout()` in an event handler to execute some logic after a time delay, wrap your code in `$A.getCallback()`. You don't need to use `$A.getCallback()` if your code is executed as part of the framework's call stack; for example, your code is handling an event or in the callback for a server-side controller action.

Client Payload Data Limit

Use `action.setParams()` to set data for an action to be passed to a server-side controller.

The framework batches the actions in the queue into one server request. The request payload includes all of the actions and their data serialized into JSON. The request payload limit is 4 MB.

IN THIS SECTION:

SEE ALSO:

Handling Events with Client-Side Controllers
Passing Data to an Apex Controller
Queueing of Server-Side Actions
Action States
Checking Component Validity

Action States

Call a server-side controller action from a client-side controller. The action can have different states during processing.

The possible action states are:

NEW
The action was created but is not in progress yet

RUNNING
The action is in progress

SUCCESS
The action executed successfully

ERROR
The server returned an error

INCOMPLETE
The server didn't return a response. The server might be down or the client might be offline. The framework guarantees that an action's callback is always invoked as long as the component is valid. If the socket to the server is never successfully opened, or closes abruptly, or any other network error occurs, the XHR resolves and the callback is invoked with state equal to `INCOMPLETE`.

ABORTED
The action was aborted. This action state is deprecated. A callback for an aborted action is executed only if you explicitly add a handler for it.

SEE ALSO:

Calling a Server-Side Action

Passing Data to an Apex Controller

Use `action.setParams()` in JavaScript to set data to pass to an Apex controller.

This example sets the value of the `firstName` argument on an Apex controller's `serverEcho` method based on the `firstName` attribute value.

```
var action = cmp.get("c.serverEcho");
action.setParams({ firstName : "Jennifer" });
```

The request payload includes the action data serialized into JSON.

Here's the Apex controller method.

```
@AuraEnabled
public static String serverEcho(String firstName) {
    return ('Hello from the server, ' + firstName);
}
```

The framework deserializes the action data into the appropriate Apex type. In this example, we have a `String` parameter called `firstName`.

Example with Different Data Types

Let's look at an application that sends data of various types to an Apex controller. Each button starts the sequence of passing data of a different type.

```
<!-- actionParamTypes.app -->
<aura:application controller="ApexParamTypesController">
    <lightning:button label="putboolean" onclick="{!c.putbooleanc}"/>
    <lightning:button label="putint" onclick="{!c.putintc}"/>
    <lightning:button label="putlong" onclick="{!c.putlongc}"/>
    <lightning:button label="putdecimal" onclick="{!c.putdecimalc}"/>
    <lightning:button label="putdouble" onclick="{!c.putdoublec}"/>
    <lightning:button label="putstring" onclick="{!c.putstringc}"/>
    <lightning:button label="putobject" onclick="{!c.putobjectc}"/>
    <lightning:button label="putblob" onclick="{!c.putblobc}"/>
    <lightning:button label="putdate" onclick="{!c.putdatec}"/>
    <lightning:button label="putdatetime" onclick="{!c.putdatetimec}"/>
    <lightning:button label="puttime" onclick="{!c.puttimec}"/>
    <lightning:button label="putlistoflistoflistofstring"
onclick="{!c.putlistoflistoflistofstringc}"/>
    <lightning:button label="putmapofstring" onclick="{!c.putmapofstringc}"/>
    <lightning:button label="putcustomclass" onclick="{!c.putcustomclassc}"/>
</aura:application>
```

Here's the application's JavaScript controller. Each action calls the helper's `putdatatype` method, which queues up the actions to send to the Apex controller. The method has three parameters:

1. The component
2. The Apex method name
3. The data to pass to the Apex method

```
// actionParamTypesController.js
({
    putbooleanc : function(component, event, helper) {
        helper.putdatatype(component, "c.pboolean", true);
    },
    putintc : function(component, event, helper) {
        helper.putdatatype(component, "c.pint", 10);
    },
    putlongc : function(component, event, helper) {
        helper.putdatatype(component, "c.plong", 2147483648);
    },
    putdecimalc : function(component, event, helper) {
        helper.putdatatype(component, "c.pdecimal", 10.80);
    },
    putdoublec : function(component, event, helper) {
        helper.putdatatype(component, "c.pdouble", 10.80);
    },
    putstringc : function(component, event, helper) {
        helper.putdatatype(component, "c.pstring", "hello!");
    },
    putobjectc : function(component, event, helper) {
        helper.putdatatype(component, "c.pobject", true);
    },
    putblobc : function(component, event, helper) {
        helper.putdatatype(component, "c.pblob", "some blob as string");
    },
    // Date value is in ISO 8601 date format
    putdatec : function(component, event, helper) {
        helper.putdatatype(component, "c.pdate", "2020-01-31");
    },
    // Datetime value is in ISO 8601 datetime format
    putdatetimec : function(component, event, helper) {
        helper.putdatatype(component, "c.pdatetime", "2020-01-31T15:08:16.000Z");
    },
    // Set time in milliseconds.
    // You can use (new Date()).getTime() to set the milliseconds
    puttimec : function(component, event, helper) {
        helper.putdatatype(component, "c.ptime", 3723004);
        //helper.putdatatype(component, "c.ptime", (new Date()).getTime());
    },
    putlistoflistoflistofstringc : function(component, event, helper) {
        helper.putdatatype(component, "c.plistoflistoflistofstring",
[[['a','b'],['c','d']],[['e','f']]]);
    },
    putmapofstringc : function(component, event, helper) {
        helper.putdatatype(component, "c.pmapofstring", {k1: 'v1'});
    },
    putcustomclassc : function(component, event, helper) {
        helper.putdatatype(component, "c.pcustomclass", {
            s: 'my string',
            i: 10,
            l: ['list value 1','list value 2'],
            m: {k1: 'map value'},
            os: {b: true}
        });
    },
})
```

The helper has a utility method to send the data to an Apex controller.

```
// actionParamTypesHelper.js
({
    putdatatype : function(component, actionName, val) {
        var action = component.get(actionName);
        action.setParams({ v : val });
        action.setCallback(this, function(response) {
            console.log(response.getReturnValue());
        });
        $A.enqueueAction(action);
    }
})
```

Here's the Apex controller.

```
public class ApexParamTypesController {
    @AuraEnabled
    public static Boolean pboolean(Boolean v){
        System.debug(v);
        return v;
    }
    @AuraEnabled
    public static Integer pint(Integer v){
        System.debug(v+v);
        return v;
    }
    @AuraEnabled
    public static Long plong(Long v){
        System.debug(v);
        return v;
    }
    @AuraEnabled
    public static Decimal pdecimal(Decimal v){
        System.debug(v);
        return v;
    }
    @AuraEnabled
    public static Double pdouble(Double v){
        System.debug(v);
        return v;
    }
    @AuraEnabled
    public static String pstring(String v){
        System.debug(v.capitalize());
        return v;
    }
    @AuraEnabled
    public static Object pobject(Object v){
        System.debug(v);
        return v;
    }
    @AuraEnabled
    public static Blob pblob(Blob v){
          System.debug(v.toString());
        return v;
    }
    @AuraEnabled
    public static Date pdate(Date v){
          System.debug(v);
        return v;
    }
    @AuraEnabled
    public static DateTime pdatetime(DateTime v){
        System.debug(v);
        return v;
    }
    @AuraEnabled
    public static Time ptime(Time v){
        System.debug(v);
        return v;
    }
    @AuraEnabled
   public static List<List<List<String>>> plistoflistoflistofstring(List<List<List<String>>>
 v){
        System.debug(v);
        return v;
    }
     @AuraEnabled
    public static Map<String, String> pmapofstring(Map<String, String> v){
        System.debug(v);
        return v;
    }
    @AuraEnabled
    public static MyCustomApexClass pcustomclass(MyCustomApexClass v){
        System.debug(v);
        return v;
    }
}
```

The `pcustomclass()` Apex method has a parameter that's a custom Apex type, `MyCustomApexClass`. Each property in the Apex class must have an `@AuraEnabled` annotation, as well as a getter and setter.

```
public class MyCustomApexClass {
    @AuraEnabled
    public String s {get; set;}
    @AuraEnabled
    public Integer i {get; set;}
    @AuraEnabled
    public List<String> l {get; set;}
    @AuraEnabled
    public Map <String, String> m {get; set;}
    @AuraEnabled
    public MyOtherCustomApexClass os {get; set;}
}
```

The `MyCustomApexClass` Apex class has a property with a type of another custom Apex class, `MyOtherCustomApexClass`.

```
public class MyOtherCustomApexClass {
    @AuraEnabled
    public Boolean b {get; set;}
}
```

Note: When Lightning Web Security is enabled, you can't use an Apex inner class as a parameter or return value for an Apex method that's called by an Aura component.

SEE ALSO:

Queueing of Server-Side Actions

Apex Server-Side Controller Overview

Returning Data from an Apex Server-Side Controller

Return results from a server-side controller to a client-side controller using the `return` statement. Results data must be serializable into JSON format.

Return data types can be any of the following.

- Simple—String, Integer, and so on. See Basic Types for details.
- sObject—standard and custom sObjects are both supported. See Standard and Custom Object Types.
- Apex—an instance of an Apex class. See Custom Apex Class Types. You can't use an Apex inner class as a return value for an Apex method that's called by an Aura component.
- Collection—a collection of any of the other types. See Collection Types.

Returning Apex Objects

Here's an example of a controller that returns a collection of custom Apex objects.

```
public with sharing class SimpleAccountController {

    @AuraEnabled
    public static List<SimpleAccount> getAccounts() {

        // Perform isAccessible() check here

        // SimpleAccount is a simple "wrapper" Apex class for transport
        List<SimpleAccount> simpleAccounts = new List<SimpleAccount>();

        List<Account> accounts = [SELECT Id, Name, Phone FROM Account LIMIT 5];
        for (Account acct : accounts) {
            simpleAccounts.add(new SimpleAccount(acct.Id, acct.Name, acct.Phone));
        }

        return simpleAccounts;
    }
}
```

When an instance of an Apex class is returned from a server-side action, the framework serializes the return data into JSON format. Only the values of `public` instance properties and methods annotated with `@AuraEnabled` are serialized and returned.

These Apex data types are serialized from `@AuraEnabled` properties and methods. They are supported as Aura component attributes.

- Primitive types except for BLOB
- Objects
- sObjects
- Lists and Maps if they hold elements of a supported type

For example, here's a wrapper Apex class that contains a few details for an account record. This class is used to package a few details of an account record in a serializable format.

```
public class SimpleAccount {

    @AuraEnabled public String Id { get; set; }
    @AuraEnabled public String Name { get; set; }
    public String Phone { get; set; }
    // Trivial constructor, for server-side Apex -> client-side JavaScript
    public SimpleAccount(String id, String name, String phone) {
        this.Id = id;
        this.Name = name;
        this.Phone = phone;
    }

    // Default, no-arg constructor, for client-side -> server-side
    public SimpleAccount() {}

}
```

When returned from a remote Apex controller action, the Id and Name properties are defined on the client-side. However, because it doesn't have the `@AuraEnabled` annotation, the Phone property isn't serialized on the server side, and isn't returned as part of the result data.

Note: Standard Apex limits, such as the maximum number of records retrieved by SOQL queries, apply when returning data from a server-side controller. See Execution Governors and Limits in the *Apex Developer Guide*.

SEE ALSO:

AuraEnabled Annotation

Custom Apex Class Types

Calling a Server-Side Action

Returning Errors from an Apex Server-Side Controller

Create and throw a `System.AuraHandledException` from your Apex controller to return a custom error message to a JavaScript controller.

Errors happen. Sometimes they're expected, such as invalid input from a user, or a duplicate record in a database. Sometimes they're unexpected, such as... Well, if you've been programming for any length of time, you know that the range of unexpected errors is nearly infinite.

When your Apex controller code experiences an error, two things can happen. You can use a catch block and handle the error in Apex. Otherwise, the error is passed back in the controller's response.

If you handle the error in Apex, you again have two ways you can go. You can process the error in a catch block, perhaps recovering from it, and return a normal response to the client. Or, you can create and throw an `AuraHandledException`.

The benefit of throwing `AuraHandledException`, instead of letting a system exception be returned, is that you have a chance to handle the exception more gracefully in your JavaScript controller code. System exceptions have important details stripped out for security purposes, and result in the dreaded "An internal server error has occurred…" message. Nobody likes that. When you use an `AuraHandledException` you have an opportunity to add some detail back into the response returned to your client-side code. More importantly, you can choose a better message to show your users.

Here's an example of creating and throwing an `AuraHandledException` in response to bad input. However, the real benefit of using `AuraHandledException` comes when you use it in response to a system exception. For example, throw an `AuraHandledException` in response to catching a DML exception, instead of allowing the DML exception to propagate to your client component code.

```
public with sharing class SimpleErrorController {

    static final List<String> BAD_WORDS = new List<String> {
        'bad',
        'words',
        'here'
    };

    @AuraEnabled
    public static String helloOrThrowAnError(String name) {

        // Make sure we're not seeing something naughty
        for(String badWordStem : BAD_WORDS) {
            if(name.containsIgnoreCase(badWordStem)) {
                // How rude! Gracefully return an error...
                throw new AuraHandledException('NSFW name detected.');
            }
        }

        // No bad word found, so...
        return ('Hello ' + name + '!');
    }

}
```

This JavaScript controller code handles the `AuraHandledException` thrown by the Apex controller.

```
({
    "callServer" : function(cmp) {
        var action = cmp.get("c.helloOrThrowAnError");
        action.setParams({ name : "bad" });

        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                console.log("From server: " + response.getReturnValue());
            }
            else if (state === "INCOMPLETE") {
                // do something
            }
            else if (state === "ERROR") {
                var errors = response.getError();
                if (errors) {
                    if (errors[0] && errors[0].message) {
                        // log the error passed in to AuraHandledException
                        console.log("Error message: " +
                                 errors[0].message);
                    }
                } else {
                    console.log("Unknown error");
                }
            }
        });

        $A.enqueueAction(action);
    }
})
```

When an Apex controller throws an `AuraHandledException`, the response state in the JavaScript controller is set to `ERROR` and you can get the error message by processing `response.getError()`.

This example simply logs the error to the console. To display an error prompt in the UI, use the `lightning:notificationsLibrary` component.

SEE ALSO:

Salesforce Developers Blog: Error Handling Best Practices for Lightning and Apex

Queueing of Server-Side Actions

The framework queues up actions before sending them to the server. This mechanism is largely transparent to you when you're writing code but it enables the framework to minimize network traffic by batching multiple actions into one request (XHR).

The batching of actions is also known as *boxcar'ing*, similar to a train that couples boxcars together.

Multiple actions sent in the same boxcar are processed in one transaction.

The framework uses a stack to track the actions to send to the server. When the browser finishes processing events and JavaScript on the client, the enqueued actions on the stack are sent to the server in a batch.

Tip: If your action isn't executing, make sure that you're not executing code outside the framework's normal rendering lifecycle. For example, if you use `window.setTimeout()` in an event handler to execute some logic after a time delay, wrap your code in `$A.getCallback()`.

There are some properties that you can set on an action to influence how the framework manages the action while it's in the queue waiting to be sent to the server. For more information, see:

- Foreground and Background Actions on page 439
- Storable Actions on page 441
- Abortable Actions on page 444

Action Limit in a Boxcar Request

The framework returns a 413 HTTP response status code if there are more than 250 actions in a boxcar request. If a user sees this rare error, consider redesigning your custom component to follow best practices and reduce the number of actions in a request.

SEE ALSO:

Modifying Components Outside the Framework Lifecycle

Foreground and Background Actions

Foreground actions are the default. An action can be marked as a background action. This is useful when you want your app to remain responsive to a user while it executes a low priority, long-running action. A rough guideline is to use a background action if it takes more than five seconds for the response to return from the server.

Batching of Actions

Multiple queued foreground actions are batched in a single request (XHR) to minimize network traffic. The batching of actions is also known as *boxcar'ing*, similar to a train that couples boxcars together.

The server sends the XHR response to the client when all actions have been processed on the server. If a long-running action is in the boxcar, the XHR response is held until that long-running action completes. Marking an action as background results in that action being sent separately from any foreground actions. The separate transmission ensures that the background action doesn't impact the response time of the foreground actions.

When the server-side actions in the queue are executed, the foreground actions execute first and then the background actions execute. Background actions run in parallel with foreground actions and responses of foreground and background actions may come back in either order.

We don't make any guarantees for the order of execution of action callbacks. XHR responses may return in a different order than the order in which the XHR requests were sent due to server processing time.

Note: Don't rely on each background action being sent in its own request as that behavior isn't guaranteed and it can lead to performance issues. Remember that the motivation for background actions is to isolate long-running requests into a separate request to avoid slowing the response for foreground actions.

Multiple actions sent in the same boxcar are processed in one transaction. If you see an error for "uncommitted work pending", it's possible that a later action can't be completed due to uncommitted work for an earlier action in the same transaction. For example, if the first action updates a record, an Apex callout in a second action can't be completed due to the uncommitted work from the first action. If two actions must be executed sequentially, the component must orchestrate the ordering. The component can enqueue the first action. In the first action's callback, the component can then enqueue the second action.

Framework-Managed Request Throttling

The framework throttles foreground and background requests separately. This means that the framework can control the number of foreground requests and the number of background actions running at any time. The framework automatically throttles requests and it's not user controlled. The framework manages the number of foreground and background XHRs, which varies depending on available resources.

Even with separate throttling, background actions might affect performance in some conditions, such as an excessive number of requests to the server.

Setting Background Actions

To set an action as a background action, call the `setBackground()` method on the action object in JavaScript.

```
// set up the server-action action
var action = cmp.get("c.serverEcho");
// optionally set actions params
//action.setParams({ firstName : cmp.get("v.firstName") });
// set as a background action
action.setBackground();
```

Note: A background action can't be set back to a foreground action. In other words, calling `setBackground` to set it to `false` will have no effect.

SEE ALSO:

Queueing of Server-Side Actions

Calling a Server-Side Action

Storable Actions

Enhance your component's performance by marking actions as storable (cacheable) to quickly show cached data from client-side storage without waiting for a server trip. If the cached data is stale, the framework retrieves the latest data from the server. Caching is especially beneficial for users on high latency, slow, or unreliable connections such as 3G networks.

Warning:

- A storable action might result in no call to the server. Never mark as storable an action that updates or deletes data.
- For storable actions in the cache, the framework returns the cached response immediately and also refreshes the data if it's stale. Therefore, storable actions might have their callbacks invoked more than once: first with cached data, then with updated data from the server.

Most server requests are read-only and idempotent, which means that a request can be repeated or retried as often as necessary without causing data changes. The responses to idempotent actions can be cached and quickly reused for subsequent identical actions. For storable actions, the key for determining an identical action is a combination of:

- Apex controller name
- Method name
- Method parameter values

Note: Client-side storage is automatically configured in Lightning Experience and the Salesforce mobile app. A component shouldn't assume a cache duration because it may change as we optimize the platform.

Marking an Action as Storable

To cache data returned from an Apex method for any component with an API version of 44.0 or higher, you must annotate the Apex method with `@AuraEnabled(cacheable=true)`. For example:

```
@AuraEnabled(cacheable=true)
public static Account getAccount(Id accountId) {
    // your code here
}
```

Prior to API version 44.0, to cache data returned from an Apex method, you had to call `setStorable()` in JavaScript code on every action that called the Apex method. For API version of 44.0 or higher, you can mark the Apex method as storable (cacheable) and get rid of any `setStorable()` calls in JavaScript code. The Apex annotation approach is better because it centralizes your caching notation for a method in the Apex class.

Call `setStorable()` on an action in JavaScript code, as follows.

```
action.setStorable();
```

The `setStorable` function takes an optional argument, which is a configuration map of key-value pairs representing the storage options and values to set. You can only set the following property:

`ignoreExisting`

Set to `true` to bypass the cache. The default value is `false`.

This property is useful when you know that any cached data is invalid, such as after a record modification. This property should be used rarely because it explicitly defeats caching.

To set the storage options for the action response, pass this configuration map into `setStorable(`***`configObj`***`)`.

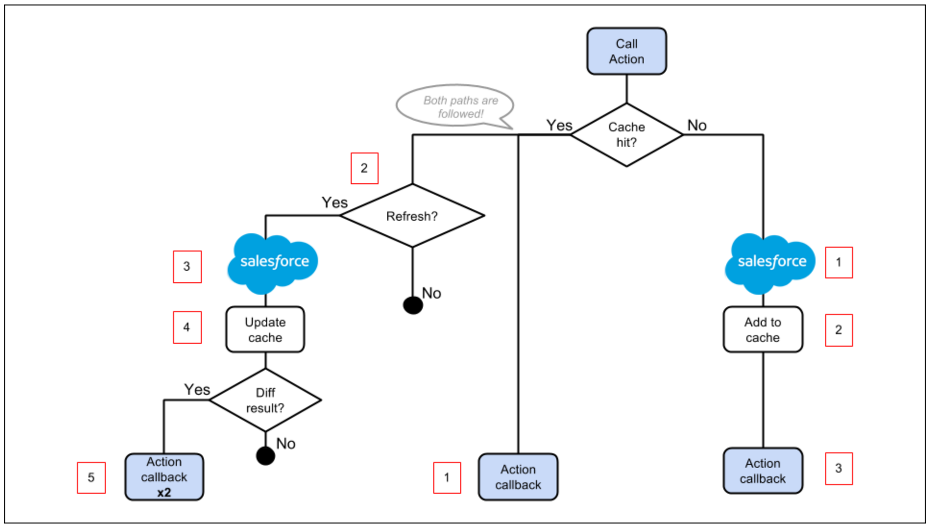
IN THIS SECTION:

Lifecycle of Storable Actions

This image describes the sequence of callback execution for storable actions.

Note: An action might have its callback invoked more than once:

- First with the cached response, if it's in storage.
- Second with updated data from the server, if the stored response has exceeded the time to refresh entries.



Cache Miss

If the action is not a cache hit as it doesn't match a storage entry:

1. The action is sent to the server-side controller.
2. If the response is `SUCCESS`, the response is added to storage.
3. The callback in the client-side controller is executed.

Cache Hit

If the action is a cache hit as it matches a storage entry:

1. The callback in the client-side controller is executed with the cached action response.

2. If the response has been cached for longer than the refresh time, the storage entry is refreshed.

   When an application enables storable actions, a refresh time is configured. The refresh time is the duration in seconds before an entry is refreshed in storage. The refresh time is automatically configured in Lightning Experience and the Salesforce mobile app.

3. The action is sent to the server-side controller.
4. If the response is `SUCCESS`, the response is added to storage.
5. If the refreshed response is different from the cached response, the callback in the client-side controller is executed for a second time.

SEE ALSO:

Storable Actions

Enable Storable Actions in an Application

Enable Storable Actions in an Application

To use storable actions in a standalone app (`.app` resource), you must configure client-side storage for cached action responses.

Note: Client-side storage is automatically configured in Lightning Experience and the Salesforce mobile app. A component shouldn't assume a cache duration because it may change as we optimize the platform.

To configure client-side storage for your standalone app, use `<auraStorage:init>` in the `auraPreInitBlock` attribute of your application's template. For example:

```
<aura:component isTemplate="true" extends="aura:template">
    <aura:set attribute="auraPreInitBlock">
        <auraStorage:init
          name="actions"
          persistent="false"
          secure="true"
          maxSize="1024"
          defaultExpiration="900"
          defaultAutoRefreshInterval="30" />
    </aura:set>
</aura:component>
```

`name`
The storage name must be `actions`. Storable actions are the only currently supported type of storage.

`persistent`
Set to `true` to preserve cached data between user sessions in the browser.

`secure`
Set to `true` to encrypt cached data.

`maxsize`
The maximum size in KB of the storage.

`defaultExpiration`
The duration in seconds that an entry is retained in storage.

`defaultAutoRefreshInterval`
The duration in seconds before an entry is refreshed in storage.

Storable actions use the Storage Service. The Storage Service supports multiple implementations of storage and selects an adapter at runtime based on browser support and specified characteristics of persistence and security.

SEE ALSO:

Storage Service Adapters

Storage Service Adapters

The Storage Service supports multiple implementations of storage and selects an adapter at runtime based on browser support and specified characteristics of persistence and security. Storage can be persistent and secure. With persistent storage, cached data is preserved between user sessions in the browser. With secure storage, cached data is encrypted.

Storage Adapter Name	Persistent	Secure
IndexedDB	`true`	`false`
Memory	`false`	`true`

IndexedDB
(Persistent but not secure) Provides access to an API for client-side storage and search of structured data. For more information, see the Indexed Database API.

Memory
(Not persistent but secure) Provides access to JavaScript memory for caching data. The stored cache persists only per browser page. Browsing to a new page resets the cache.

The Storage Service selects a storage adapter on your behalf that matches the persistent and secure options you specify when initializing the service. For example, if you request a persistent and insecure storage service, the Storage Service returns the IndexedDB storage if the browser supports it.

Abortable Actions

Mark an action as abortable to make it potentially abortable while it's queued to be sent to the server. An abortable action in the queue is not sent to the server if the component that created the action is no longer valid, that is `cmp.isValid() == false`. A component is automatically destroyed and marked invalid by the framework when it is unrendered.

Note: We recommend that you only use abortable actions for read-only operations as they are not guaranteed to be sent to the server.

An abortable action is sent to the server and executed normally unless the component that created the action is invalid before the action is sent to the server.

A non-abortable action is always sent to the server and can't be aborted in the queue.

If an action response returns from the server and the associated component is now invalid, the logic has been executed on the server but the action callback isn’t executed. This is true whether or not the action is marked as abortable.

Marking an Action as Abortable

Mark a server-side action as abortable by using the `setAbortable()` method on the `Action` object in JavaScript. For example:

```
var action = cmp.get("c.serverEcho");
action.setAbortable();
```

SEE ALSO:

- Creating Server-Side Logic with Controllers
- Queueing of Server-Side Actions
- Calling a Server-Side Action

Testing Your Apex Code

Before you can upload a managed package, you must write and execute tests for your Apex code to meet minimum code coverage requirements. Also, all tests must run without errors when you upload your package to AppExchange.

To package your application and components that depend on Apex code, the following must be true.

- Unit tests must cover at least 75% of your Apex code, and all of those tests must complete successfully.

  Note the following.

  - When deploying Apex to a production organization, each unit test in your organization namespace is executed by default.
  - Calls to `System.debug` aren't counted as part of Apex code coverage.
  - Test methods and test classes aren't counted as part of Apex code coverage.
  - While only 75% of your Apex code must be covered by tests, don't focus on the percentage of code that is covered. Instead, make sure that every use case of your application is covered, including positive and negative cases, as well as bulk and single records. This approach ensures that 75% or more of your code is covered by unit tests.
- Every trigger must have some test coverage.
- All classes and triggers must compile successfully.

This sample shows an Apex test class for a custom object that's wired up to a component.

```
@isTest
class TestExpenseController {
    static testMethod void test() {
        //Create new expense and insert it into the database
        Expense__c exp = new Expense__c(name='My New Expense',
                             amount__c=20, client__c='ABC',
                             reimbursed__c=false, date__c=null);
         ExpenseController.saveExpense(exp);

        //Assert the name field and saved expense
        System.assertEquals('My New Expense',
                           ExpenseController.getExpenses()[0].Name,
                          'Name does not match');
        System.assertEquals(exp, ExpenseController.saveExpense(exp));
    }
}
```

Note: Apex classes must be manually added to your package.

For more information on distributing Apex code, see Debugging, Testing, and Deploying Apex in the *Apex Developer Guide*.

SEE ALSO:

Distributing Applications and Components

Making API Calls from Apex

Make API calls from an Apex controller. You can't make Salesforce API calls from JavaScript code.

For security reasons, the Lightning Component framework places restrictions on making API calls from JavaScript code. To call third-party APIs from your component's JavaScript code, add the API endpoint as a CSP Trusted Site.

To call Salesforce APIs, make the API calls from your component's Apex controller. Use a named credential to authenticate to Salesforce.

Note: By security policy, sessions created by Lightning components aren't enabled for API access. This prevents even your Apex code from making API calls to Salesforce. Using a named credential for specific API calls allows you to carefully and selectively bypass this security restriction.

The restrictions on API-enabled sessions aren't accidental. Carefully review any code that uses a named credential to ensure you're not creating a vulnerability.

For information about making API calls from Apex, see the *Apex Developer Guide*.

SEE ALSO:

Apex Developer Guide: Named Credentials as Callout Endpoints

Making API Calls from Components

Make Long-Running Callouts with Continuations

Use the `Continuation` class in Apex to make a long-running request to an external web service. Process the response in a callback method. Continuations are the preferred way to manage callouts because they can provide substantial improvements to the user experience.

Using continuations has some advantages, including the capability to make callouts in parallel.

The framework queues up actions before sending them to the server. This mechanism is largely transparent to you when you're writing code but it enables the framework to minimize network traffic by batching multiple actions into one request (XHR). The batching of actions is also known as boxcar'ing, similar to a train that couples boxcars together. Since continuations can be long-running requests, the framework essentially treats continuations as background actions. Continuations aren't boxcar'ed with other requests so they don't block other actions while they are running.

An asynchronous callout made with a continuation doesn't count toward the Apex limit of synchronous requests that last longer than five seconds. Since Winter '20, all callouts are excluded from the long-running request limit so continuations no longer offer an advantage for working with limits compared to regular callouts. However, we recommend using continuations to manage callouts due to the improved user experience.

IN THIS SECTION:

Work with a Continuation in an Apex Class

To work with a continuation in an Apex class, use the Apex `Continuation` object.

@AuraEnabled Annotations for Continuations

Continuations use the `@AuraEnabled` annotation for Apex code. Here are the rules for usage.

Aura Component Continuations Example
Here's the markup for a component with a button that starts the process of calling a continuation.

Continuation-Specific Limits
Because continuations can lead to multiple long-running actions, there are some limits on their usage.

SEE ALSO:

Queueing of Server-Side Actions

Apex Reference Guide: Continuation Class

Apex Developer Guide: Named Credentials as Callout Endpoints

Work with a Continuation in an Apex Class

To work with a continuation in an Apex class, use the Apex `Continuation` object.

1. Before you can call an external service, you must add the remote site to a list of authorized remote sites in the Salesforce user interface. From **Setup**, in the **Quick Find** box, enter *`Remote Site Settings`*. Select **Remote Site Settings**, and then click **New Remote Site**. Add the callout URL corresponding to `LONG_RUNNING_SERVICE_URL` in the Apex Class Continuation example below.

   If the callout specifies a named credential as the endpoint, you don't need to configure remote site settings. A named credential specifies the URL of a callout endpoint and its required authentication parameters in one definition. In your code, specify the named credential URL instead of the long-running service URL.

2. To make a long-running callout, define an Apex method that returns a `Continuation` object. (Don't worry about the attributes of the `@AuraEnabled` annotation yet. We explain that soon.)

```
@AuraEnabled(continuation=true cacheable=true)
public static Object startRequest() {
    // Create continuation. Argument is timeout in seconds.
    Continuation con = new Continuation(40);
    // more to come here
    return con;
}
```

3. Set an Apex callback method to be invoked after the callout completes in the `continuationMethod` property of the `Continuation` object. In this example, the callback method is `processResponse`. The callback method must be in the same Apex class.

```
con.continuationMethod='processResponse';
```

4. Set the endpoint for a callout by adding an `HttpRequest` object to the `Continuation` object. A single `Continuation` object can contain a maximum of three callouts. Each callout must have a remote site or named credential defined in Setup.

```
HttpRequest req = new HttpRequest();
req.setMethod('GET');
req.setEndpoint(LONG_RUNNING_SERVICE_URL);
con.addHttpRequest(req);
```

5. Set data to pass to the callback method in the `state` property of the `Continuation` object. The `state` property has an `Object` type so you can pass in any data type that's supported in Apex.

```
con.state='Hello, World!';
```

6. Code the logic in the Apex callback. When all the callouts set in the `Continuation` object have completed, the Apex callback method, `processResponse`, is invoked. The callback method has two parameters that you can access.

   ```
   public static Object processResponse(List<String> labels, Object state)
   ```

   a. `labels`—A list of labels, one for each request in the continuation. These labels are automatically created.
   b. `state`—The state that you set in the `state` property in your `Continuation` object.

7. Get the response for each request in the continuation. For example:

   ```
   HttpResponse response = Continuation.getResponse(labels[0]);
   ```

8. Return the results to the JavaScript controller.

Complete Apex Class Example with Continuation

Here's a complete Apex class that ties together all the earlier steps.

```
public with sharing class SampleContinuationClass {
    // Callout endpoint as a named credential URL
    // or, as shown here, as the long-running service URL
    private static final String LONG_RUNNING_SERVICE_URL =
        '<insert your callout URL here>';

    // Action method
    @AuraEnabled(continuation=true cacheable=true)
    public static Object startRequest() {
      // Create continuation. Argument is timeout in seconds.
      Continuation con = new Continuation(40);
      // Set callback method
      con.continuationMethod='processResponse';
      // Set state
      con.state='Hello, World!';
      // Create callout request
      HttpRequest req = new HttpRequest();
      req.setMethod('GET');
      req.setEndpoint(LONG_RUNNING_SERVICE_URL);
      // Add callout request to continuation
      con.addHttpRequest(req);
      // Return the continuation
      return con;
    }

    // Callback method
    @AuraEnabled(cacheable=true)
    public static Object processResponse(List<String> labels, Object state) {
      // Get the response by using the unique label
      HttpResponse response = Continuation.getResponse(labels[0]);
      // Set the result variable
      String result = response.getBody();
      return result;
```

```
    }
}
```

SEE ALSO:

Make Long-Running Callouts with Continuations

`@AuraEnabled` Annotations for Continuations

Continuations use the `@AuraEnabled` annotation for Apex code. Here are the rules for usage.

`@AuraEnabled(continuation=true)`

An Apex controller method that returns a continuation must be annotated with `@AuraEnabled(continuation=true)`.

`@AuraEnabled(continuation=true cacheable=true)`

To cache the result of a continuation action, set `cacheable=true` on the annotation for the Apex callback method.

Note: There's a space, **not a comma**, between `continuation=true cacheable=true`.

Caching Considerations

It's best practice to set `cacheable=true` on all methods involved in the continuation chain, including the method that returns a `Continuation` object. The `cacheable=true` setting is available for API version 44.0 and higher. Before API version 44.0, to cache data returned from an Apex method, you had to call `setStorable()` in JavaScript code on every action that called the Apex method.

In this example, the Apex method that returns the continuation, `startRequest()`, and the callback, `processResponse()`, both contain `cacheable=true` in their `@AuraEnabled` annotation.

```
// Action method
@AuraEnabled(continuation=true cacheable=true)
public static Object startRequest() { }

// Callback method
@AuraEnabled(cacheable=true)
public static Object processResponse(List<String> labels,
  Object state) { }
```

Here's a table that summarizes the behavior with different settings of the `cacheable` attribute in `@AuraEnabled`.

Method Returning Continuation Object Annotated with `cacheable=true`	Callback Method Annotated with `cacheable=true`	Valid?	Action can use `setStorable()` in JavaScript?	Is Action Response Cached on Client?
Yes	Yes	Yes	Yes	Yes
Yes	No	No (throws an exception)	N/A	N/A
No	Yes	Yes	No (all methods must have `cacheable=true`)	Yes

Method Returning Continuation Object Annotated with `cacheable=true`	Callback Method Annotated with `cacheable=true`	Valid?	Action can use `setStorable()` in JavaScript?	Is Action Response Cached on Client?
No	No	Yes	• No (API version 44.0 and higher) • Yes (43.0 and lower)	• No (API version 44.0 and higher) • Yes (43.0 and lower)

SEE ALSO:

Make Long-Running Callouts with Continuations

AuraEnabled Annotation

Aura Component Continuations Example

Here's the markup for a component with a button that starts the process of calling a continuation.

The component is wired to the Apex class that uses a continuation by setting the controller attribute in the `<aura:component>` tag.

```
<aura:component controller="SampleContinuationClass">
    <lightning:button label="Call Continuation" onclick="{!c.callContinuation}"/>
</aura:component>
```

Here's the component's JavaScript controller. The code calls the `startRequest` Apex method that uses a `Continuation` object. The `response.getReturnValue()` value for a successful response in the JavScript controller corresponds to the value returned by the Apex callback method defined in the `Continuation` object.

```
({
    callContinuation : function(cmp) {
        var action = cmp.get("c.startRequest");
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                console.log("From server: "
                  + response.getReturnValue()
                  + '\n' + JSON.stringify(response.getReturnValue()));
            }
            else if (state === "INCOMPLETE") {
                alert("Continuation action is INCOMPLETE");
            }
            else if (state === "ERROR") {
                var errors = response.getError();
                if (errors) {
                    if (errors[0] && errors[0].message) {
                        console.log("Error message: " +
                                 errors[0].message);
                    }
                } else {
                    console.log("Unknown error");
                }
            }
        });
        // Enqueue action that returns a continuation
        $A.enqueueAction(action);
    }
})
```

This JavaScript controller code is similar to any other component that calls an Apex method.

SEE ALSO:

Make Long-Running Callouts with Continuations

Continuation-Specific Limits

Because continuations can lead to multiple long-running actions, there are some limits on their usage.

The limits for using continuations in Apex are listed in the Apex Reference Guide.

Here are a few more limits specific to usage in Aura components.

Up to three callouts per continuation

A single `Continuation` object can contain a maximum of three callouts.

Serial processing for continuation actions

The framework processes actions containing a continuation serially from the client. The previous continuation action call must have completed before the next continuation action call is made. At any time, you can have only one continuation in progress on the client.

DML operation restrictions

An Apex method that returns a `Continuation` object can't perform Data Manipulation Language (DML) operations. DML statements insert, update, merge, delete, and restore data in Salesforce. If a DML operation is performed within the continuation method, the continuation execution doesn't proceed, the transaction is rolled back, and an error is returned.

You can perform DML operations in the Apex callback method for the continuation.

SEE ALSO:

Make Long-Running Callouts with Continuations

Queueing of Server-Side Actions

Creating Components in Apex

Creating components on the server side in Apex, using the `Cmp.<myNamespace>.<myComponent>` syntax, is deprecated. Use `$A.createComponent()` in client-side JavaScript code instead.

SEE ALSO:

Dynamically Creating Components

CHAPTER 12 Testing Components

Automated tests are the best way to achieve predictable, repeatable assessments of the quality of your custom code. Writing automated tests for your custom components gives you confidence that they work as designed, and allows you to evaluate the impact of changes, such as refactoring, or of new versions of Salesforce or third-party JavaScript libraries.

Use your testing framework of choice. Here are some popular testing tools.

- Jest
- UTAM
- Jasmine
- Mocha
- Selenium
- WebdriverIO

Note: We used to recommend Lightning Testing Service (LTS) but it’s deprecated and no longer supported.

CHAPTER 13 Debugging

In this chapter ...

There are a few basic tools and techniques that can help you to debug applications.

Use Chrome DevTools to debug your client-side code.

- To open DevTools on Windows and Linux, press Control-Shift-I in your Google Chrome browser. On Mac, press Option-Command-I.
- To quickly find which line of code is failing, enable the **Pause on all exceptions** option before running your code.

To learn more about debugging JavaScript on Google Chrome, refer to the Google Chrome's DevTools website.

Disable Caching Setting During Development

Disable the secure and persistent browser caching setting during development in a sandbox or Developer Edition org to see the effect of any code changes without needing to empty the cache.

The caching setting improves page reload performance by avoiding extra round trips to the server.

Warning: Disabling secure and persistent browser caching has a significant negative performance impact on Lightning Experience. Always enable the setting in production orgs.

1. From Setup, enter *`Session`* in the `Quick Find` box, and then select **Session Settings**.
2. Deselect the checkbox for "Enable secure and persistent browser caching to improve performance".
3. Click **Save**.

SEE ALSO:

Enable Secure Browser Caching

Log Messages

To help debug your client-side code, you can write output to the JavaScript console of a web browser using `console.log()` if your browser supports it..

For instructions on using the JavaScript console, refer to the instructions for your web browser.

CHAPTER 14 Performance

In this chapter ...

There are a few settings and techniques that can help you to improve application performance.

Only enable debug mode for users who are actively debugging JavaScript. Salesforce is slower for users who have debug mode enabled.

SEE ALSO:

Salesforce Help: **Enable Debug Mode for Lightning Components**

Performance Settings

There are a few Setup settings that can help you to improve application performance.

IN THIS SECTION:

Enable Secure Browser Caching

Enable secure data caching in the browser to improve page reload performance by avoiding extra round trips to the server.

This setting is selected by default.

Warning: Disabling secure and persistent browser caching has a significant negative performance impact on Lightning Experience. Only disable in these scenarios.

- Your company's policy doesn't allow browser caching, even if the data is encrypted.
- During development in a sandbox or Developer Edition, you want to see the effect of any code changes without emptying the secure cache.

To disable secure data caching:

1. From Setup, enter `Session` in the `Quick Find` box, and then select **Session Settings**.
2. Deselect the checkbox for "Enable secure and persistent browser caching to improve performance".
3. Click **Save**.

Note: Enabling secure and persistent data caching impacts record pages in Experience Cloud. Updates on fields aren't observed immediately. To see the latest changes immediately, log out and log back in. Other users don't see the change until the cache is deleted or invalidated.

Enable the Lightning CDN to Load Applications Faster

To load Lightning Experience and other apps faster, enable the Lightning content delivery network (CDN) for your org. The Lightning CDN serves static content for the Lightning component framework.

Salesforce partners with Cloudfront to serve the static content for the Lightning Component framework over a content delivery network (CDN). CDNs are the industry standard for web applications because they provide faster and more secure content delivery. A CDN is a geographically distributed network of servers that store cached versions of web assets. To optimize page load times and site performance, a CDN efficiently delivers publicly cacheable content to users. If your company has IP range restrictions for content served from Salesforce, test thoroughly before enabling this setting.

To enable Lightning Experience, select **Enable Content Delivery Network (CDN) for Lightning Component framework** in **Setup** > **Session Settings**. This setting turns on the Lightning CDN for the static JavaScript and CSS in the Lightning Component framework at the org level. It doesn't distribute your Salesforce data or metadata in a CDN. This setting is disabled by default for orgs created before the Winter '19 release, and enabled by default for new orgs.

If you experience any issues, ask your IT department if your company's firewall blocks Cloudfront CDN content. Your IT department can ensure that static.lightning.force.com and *.static.lightning.force.com are added to any allowlist or firewall that your company operates. You can ping `static.lightning.force.com` but you can't browse directly to the root URL at `https://static.lightning.force.com`.

Important: Don't use IP addresses for network filtering because that can cause connection issues with `https://static.lightning.force.com`. IP addresses for `https://static.lightning.force.com` are dynamic and aren't maintained in Salesforce's list of allowed IP addresses.

SEE ALSO:

Salesforce Help: Options to Serve a Custom Domain

Knowledge Article: Salesforce IP Addresses and Domains to Allow

Fixing Performance Warnings

A few common performance anti-patterns in code prompt the framework to log warning messages to the browser console. Fix the warning messages to speed up your components!

The warnings display in the browser console only if you enabled debug mode.

IN THIS SECTION:

<aura:if>—Clean Unrendered Body

This warning occurs when you change the `isTrue` attribute of an `<aura:if>` tag from `true` to `false` in the same rendering cycle. The unrendered body of the `<aura:if>` must be destroyed, which is avoidable work for the framework that slows down rendering time.

<aura:iteration>—Multiple Items Set

This warning occurs when you set the `items` attribute of an `<aura:iteration>` tag multiple times in the same rendering cycle.

SEE ALSO:

Salesforce Help: **Enable Debug Mode for Lightning Components**

`<aura:if>`—Clean Unrendered Body

This warning occurs when you change the `isTrue` attribute of an `<aura:if>` tag from `true` to `false` in the same rendering cycle. The unrendered body of the `<aura:if>` must be destroyed, which is avoidable work for the framework that slows down rendering time.

Example

This component shows the anti-pattern.

```
<!--c:ifCleanUnrendered-->
<aura:component>
    <aura:attribute name="isVisible" type="boolean" default="true"/>
    <aura:handler name="init" value="{!this}" action="{!c.init}"/>
    <aura:if isTrue="{!v.isVisible}">
        <p>I am visible</p>
    </aura:if>
</aura:component>
```

Here's the component's client-side controller.

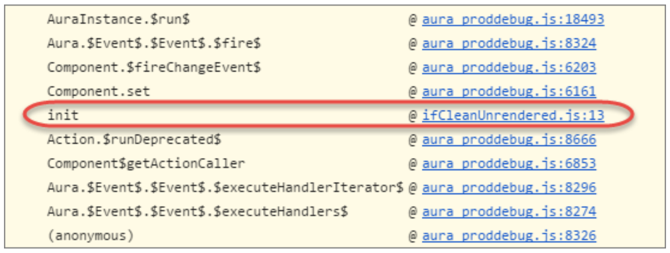
```
/* c:ifCleanUnrenderedController.js */
({
    init: function(cmp) {
        /* Some logic */
        cmp.set("v.isVisible", false); // Performance warning trigger
    }
})
```

When the component is created, the `isTrue` attribute of the `<aura:if>` tag is evaluated. The value of the `isVisible` attribute is `true` by default so the framework creates the body of the `<aura:if>` tag. After the component is created but before rendering, the `init` event is triggered.

The `init()` function in the client-side controller toggles the `isVisible` value from `true` to `false`. The `isTrue` attribute of the `<aura:if>` tag is now `false` so the framework must destroy the body of the `<aura:if>` tag. This warning displays in the browser console only if you enabled debug mode.

```
WARNING: [Performance degradation] markup://aura:if ["5:0"] in c:ifCleanUnrendered ["3:0"]
needed to clear unrendered body.
```

Click the expand button beside the warning to see a stack trace for the warning.



```
AuraInstance.$run$                                  @ aura_proddebug.js:18493
Aura.$Event$.$Event$.$fire$                         @ aura_proddebug.js:8324
Component.$fireChangeEvent$                         @ aura_proddebug.js:6203
Component.set                                       @ aura_proddebug.js:6161
init                                                @ ifCleanUnrendered.js:13
Action.$runDeprecated$                              @ aura_proddebug.js:8666
Component$getActionCaller                           @ aura_proddebug.js:6853
Aura.$Event$.$Event$.$executeHandlerIterator$       @ aura_proddebug.js:8296
Aura.$Event$.$Event$.$executeHandlers$              @ aura_proddebug.js:8274
(anonymous)                                         @ aura_proddebug.js:8326
```

Click the link for the `ifCleanUnrendered` entry in the stack trace to see the offending line of code in the Sources pane of the browser console.

How to Fix the Warning

Reverse the logic for the `isTrue` expression. Instead of setting the `isTrue` attribute to `true` by default, set it to `false`. Set the `isTrue` expression to true in the `init()` method, if needed.

Here's the fixed component:

```
<!--c:ifCleanUnrenderedFixed-->
<aura:component>
    <!-- FIX: Change default to false.
         Update isTrue expression in controller instead. -->
    <aura:attribute name="isVisible" type="boolean" default="false"/>
    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <aura:if isTrue="{!v.isVisible}">
        <p>I am visible</p>
    </aura:if>
</aura:component>
```

Here's the fixed controller:

```
/* c:ifCleanUnrenderedFixedController.js */
({
    init: function(cmp) {
        // Some logic
        // FIX: set isVisible to true if logic criteria met
        cmp.set("v.isVisible", true);
    }
})
```

SEE ALSO:

Salesforce Help: **Enable Debug Mode for Lightning Components**

`<aura:iteration>`—Multiple Items Set

This warning occurs when you set the `items` attribute of an `<aura:iteration>` tag multiple times in the same rendering cycle.

There's no easy and performant way to check if two collections are the same in JavaScript. Even if the old value of `items` is the same as the new value, the framework deletes and replaces the previously created body of the `<aura:iteration>` tag.

Example

This component shows the anti-pattern.

```
<!--c:iterationMultipleItemsSet-->
<aura:component>
    <aura:attribute name="groceries" type="List"
                  default="[ 'Eggs', 'Bacon', 'Bread' ]"/>

    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <aura:iteration items="{!v.groceries}" var="item">
        <p>{!item}</p>
    </aura:iteration>
</aura:component>
```

Here's the component's client-side controller.

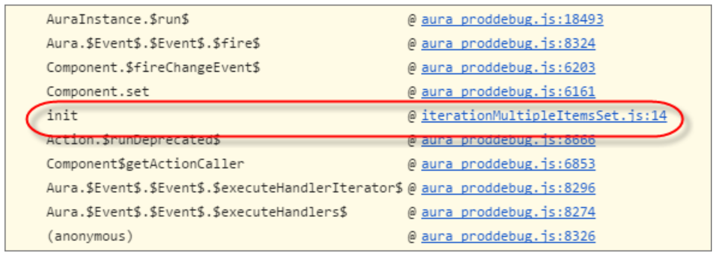
```
/* c:iterationMultipleItemsSetController.js */
({
    init: function(cmp) {
        var list = cmp.get('v.groceries');
        // Some logic
        cmp.set('v.groceries', list); // Performance warning trigger
    }
})
```

When the component is created, the `items` attribute of the `<aura:iteration>` tag is set to the default value of the `groceries` attribute. After the component is created but before rendering, the `init` event is triggered.

The `init()` function in the client-side controller sets the `groceries` attribute, which resets the `items` attribute of the `<aura:iteration>` tag. This warning displays in the browser console only if you enabled debug mode.

```
WARNING: [Performance degradation] markup://aura:iteration [id:5:0] in
c:iterationMultipleItemsSet ["3:0"]
had multiple items set in the same Aura cycle.
```

Click the expand button beside the warning to see a stack trace for the warning.



```
AuraInstance.$run$                                  @ aura_proddebug.js:18493
Aura.$Event$.$Event$.$fire$                         @ aura_proddebug.js:8324
Component.$fireChangeEvent$                         @ aura_proddebug.js:6203
Component.set                                       @ aura_proddebug.js:6161
init                                                @ iterationMultipleItemsSet.js:14
Action.$runDeprecated$                              @ aura_proddebug.js:8666
Component$getActionCaller                           @ aura_proddebug.js:6853
Aura.$Event$.$Event$.$executeHandlerIterator$       @ aura_proddebug.js:8296
Aura.$Event$.$Event$.$executeHandlers$              @ aura_proddebug.js:8274
(anonymous)                                         @ aura_proddebug.js:8326
```

Click the link for the `iterationMultipleItemsSet` entry in the stack trace to see the offending line of code in the Sources pane of the browser console.

How to Fix the Warning

Make sure that you don't modify the `items` attribute of an `<aura:iteration>` tag multiple times. The easiest solution is to remove the default value for the `groceries` attribute in the markup. Set the value for the `groceries` attribute in the controller instead.

The alternate solution is to create a second attribute whose only purpose is to store the default value. When you've completed your logic in the controller, set the `groceries` attribute.

Here's the fixed component:

```
<!--c:iterationMultipleItemsSetFixed-->
<aura:component>
    <!-- FIX: Remove the default from the attribute -->
    <aura:attribute name="groceries" type="List" />
    <!-- FIX (ALTERNATE): Create a separate attribute containing the default -->
    <aura:attribute name="groceriesDefault" type="List"
                  default="[ 'Eggs', 'Bacon', 'Bread' ]"/>

    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <aura:iteration items="{!v.groceries}" var="item">
        <p>{!item}</p>
    </aura:iteration>
</aura:component>
```

Here's the fixed controller:

```
/* c:iterationMultipleItemsSetFixedController.js */
({
    init: function(cmp) {
        // FIX (ALTERNATE) if need to set default in markup
        // use a different attribute
        // var list = cmp.get('v.groceriesDefault');
        // FIX: Set the value in code
        var list = ['Eggs', 'Bacon', 'Bread'];
        // Some logic
        cmp.set('v.groceries', list);
    }
})
```

SEE ALSO:

Salesforce Help: **Enable Debug Mode for Lightning Components**

CHAPTER 15 Reference

In this chapter ...

This section contains links to reference documentation.

Component Library

The Lightning Component Library is your hub for Lightning UI developer information, including the Component Reference with live examples, the Lightning Web Components developer guide, and tools for Lightning Web Security and Lightning Locker.

You can find the Component Library in two places: a public site and an authenticated one that's linked to your Salesforce org. In the authenticated site, the Component Reference section of the Component Library has some additional features.

Public Component Library

View the public site https://developer.salesforce.com/docs/component-librarywithout logging in to Salesforce. The Component Reference includes documentation and reference information for the base Lightning components.

Component Library for your org

View this site by logging in to your Salesforce org and navigating to `https://`*`MyDomainName`*`.my.salesforce.com/docs/component-library`. Alternatively, click **Link to your org** at the top right on the public site.

The authenticated site has additional features for the Component Reference.

- View Aura Lightning components that are unique to your org.
- View Aura Lightning components that are installed in a managed package. You can filter to view components owned by your org or installed in packages. Find the filtering options at `https://`*`MyDomainName`*`.my.salesforce.com/docs/component-library/overview/components` and expand the Filters list to find the Owners filters.

See Lightning Component Library in the Lightning Web Components Developer Guide for more information and known issues in the Component Reference.

JavaScript API Documentation

For JavaScript API documentation, see JavaScript API.

The legacy reference doc app, which used to contain the JavaScript API documentation, has been retired.

IN THIS SECTION:

Differences Between Documentation Sites
Here's a breakdown of the differences between the Component Library and the reference section of this developer guide.

Differences Between Documentation Sites

Here's a breakdown of the differences between the Component Library and the reference section of this developer guide.

The Component Library is the place to find reference information and interactive examples. The Reference section in this Developer Guide provides information on system-level tags that are not available elsewhere, as well as the JavaScript API.

	Component Library	**Reference Section in Lightning Aura Components Developer Guide**
Component documentation and code samples	✓ (Documentation tab)	
Interactive examples	✓ (Example tab)	
Lightning Design System support	✓	
Components in custom namespaces and packages	✓	
JavaScript API		✓
System tags		✓ (`aura:method`, `aura:set`, etc.)
Event documentation	✓	
System event documentation	✓	
Interface documentation	✓	

Components in custom namespaces display both global and non-global attributes and methods in the authenticated Component Library displayed in an org. Components in managed and unmanaged packages display only global attributes and methods.

System Tag Reference

System tags represent framework definitions and are not available in the Component Library.

IN THIS SECTION:

aura:application
An app is a special top-level component whose markup is in a `.app` resource.

aura:dependency
The `<aura:dependency>` tag enables you to declare dependencies, which improves their discoverability by the framework.

aura:event
An event is represented by the `aura:event` tag, which has the following attributes.

aura:interface

Interfaces determine a component's shape by defining its attributes. Implement an interface to allow a component to be used in different contexts, such as on a record page or in Lightning App Builder.

aura:method

Use `<aura:method>` to define a method as part of a component's API. This enables you to directly call a method in a component's client-side controller instead of firing and handling a component event. Using `<aura:method>` simplifies the code needed for a parent component to call a method on a child component that it contains.

aura:set

Use `<aura:set>` in markup to set the value of an attribute inherited from a component or event.

aura:application

An app is a special top-level component whose markup is in a `.app` resource.

The markup looks similar to HTML and can contain components as well as a set of supported HTML tags. The `.app` resource is a standalone entry point for the app and enables you to define the overall application layout, style sheets, and global JavaScript includes. It starts with the top-level `<aura:application>` tag, which contains optional system attributes. These system attributes tell the framework how to configure the app.

System Attribute	Type	Description
`access`	String	Indicates whether the app can be extended by another app outside of a namespace. Possible values are `public` (default), and `global`.
`controller`	String	The Apex controller class for the app. The format is `namespace.myController`.
`description`	String	A brief description of the app.
`extends`	Component	The app to be extended, if applicable. For example, `extends="namespace:yourApp"`.
`extensible`	Boolean	Indicates whether the app is extensible by another app. Defaults to `false`.
`implements`	String	A comma-separated list of interfaces that the app implements.
`template`	Component	The name of the template used to bootstrap the loading of the framework and the app. The default value is `aura:template`. You can customize the template by creating your own component that extends the default template. For example: `<aura:component extends="aura:template" ... >`
`tokens`	String	A comma-separated list of tokens bundles for the application. For example, `tokens="ns:myAppTokens"`. Tokens make it easy to ensure that your design is consistent, and even easier to update it as your design evolves. Define the token values once and reuse them throughout your application.
`useAppcache`	Boolean	Deprecated. Browser vendors have deprecated AppCache, so we followed their lead. Remove the `useAppcache` attribute in the `<aura:application>` tag of your standalone apps (`.app` resources) to avoid cross-browser support issues due to deprecation by browser vendors.

System Attribute	Type	Description
		If you don't currently set `useAppcache` in an `<aura:application>` tag, you don't have to do anything because the default value of `useAppcache` is `false`.

`aura:application` also includes a `body` attribute defined in a `<aura:attribute>` tag. Attributes usually control the output or behavior of a component, but not the configuration information in system attributes.

Attribute	Type	Description
`body`	`Component[]`	The body of the app. In markup, this is everything in the body of the tag.

SEE ALSO:

Creating Apps

Application Access Control

aura:dependency

The `<aura:dependency>` tag enables you to declare dependencies, which improves their discoverability by the framework.

The framework automatically tracks dependencies between definitions, such as components, defined in markup. This enables the framework to send the definitions to the browser. However, if a component's JavaScript code dynamically instantiates another component or fires an event that isn't directly referenced in the component's markup, use `<aura:dependency>` in the component's markup to explicitly tell the framework about the dependency. Adding the `<aura:dependency>` tag ensures that a definition, such as a component, and its dependencies are sent to the client, when needed.

For example, adding this tag to a component marks the `sampleNamespace:sampleComponent` component as a dependency.

```
<aura:dependency resource="markup://sampleNamespace:sampleComponent" />
```

Add this tag to component markup to mark the event as a dependency.

```
<aura:dependency resource="markup://force:navigateToComponent" type="EVENT"/>
```

Use the `<aura:dependency>` tag if you fire an event in JavaScript code and you're not registering the event in component markup using `<aura:registerEvent>`. Using an `<aura:registerEvent>` tag is the preferred approach.

The `<aura:dependency>` tag includes these system attributes.

System Attribute	Description
`resource`	The resource that the component depends on, such as a component or event. For example, `resource="markup://sampleNamespace:sampleComponent"` refers to the `sampleComponent` in the `sampleNamespace` namespace. Note: Using an asterisk (`*`) for wildcard matching is deprecated. Instead, add an `<aura:dependency>` tag for each resource that's not directly referenced in the component's markup. Wildcard matching can cause save validation errors when no

System Attribute	Description
	resources match. Wildcard matching can also slow page load time because it sends more definitions than needed to the client.
`type`	The type of resource that the component depends on. The default value is `COMPONENT`. Note: Using an asterisk (`*`) for wildcard matching is deprecated. Instead, add an `<aura:dependency>` tag for each resource that's not directly referenced in the component's markup. Be as selective as possible in the types of definitions that you send to the client. The most commonly used values are: • `COMPONENT` • `EVENT` • `INTERFACE` • `APPLICATION` • `MODULE`—Use this type to add a dependency for a Lightning web component Use a comma-separated list for multiple types; for example: `COMPONENT,APPLICATION`.

SEE ALSO:

Dynamically Creating Components

Fire Component Events

Fire Application Events

aura:event

An event is represented by the `aura:event` tag, which has the following attributes.

Attribute	Type	Description
`access`	String	Indicates whether the event can be extended or used outside of its own namespace. Possible values are `public` (default), and `global`.
`description`	String	A description of the event.
`extends`	Component	The event to be extended. For example, `extends="namespace:myEvent"`.
`type`	String	Required. Possible values are `COMPONENT` or `APPLICATION`.

SEE ALSO:

Communicating with Events

Event Access Control

aura:interface

Interfaces determine a component's shape by defining its attributes. Implement an interface to allow a component to be used in different contexts, such as on a record page or in Lightning App Builder.

The `aura:interface` tag has the following optional attributes.

Attribute	Type	Description
`access`	String	Indicates whether the interface can be extended or used outside of its own namespace. Possible values are `public` (default), and `global`.
`description`	String	A description of the interface.
`extends`	Component	The comma-separated list of interfaces to be extended. For example, `extends="namespace:intfB"`.

SEE ALSO:

- Interfaces
- Interface Access Control

aura:method

Use `<aura:method>` to define a method as part of a component's API. This enables you to directly call a method in a component's client-side controller instead of firing and handling a component event. Using `<aura:method>` simplifies the code needed for a parent component to call a method on a child component that it contains.

The `<aura:method>` tag has these system attributes.

Attribute	Type	Description
`name`	`String`	The method name. Use the method name to call the method in JavaScript code. For example: `cmp.sampleMethod(param1);`
`action`	`Expression`	The client-side controller action to execute. For example: `action="{!c.sampleAction}"` `sampleAction` is an action in the client-side controller. If you don't specify an `action` value, the controller action defaults to the value of the method `name`.
`access`	`String`	The access control for the method. Valid values are: • **public**—Any component in the same namespace can call the method. This is the default access level. • **global**—Any component in any namespace can call the method.

Attribute	Type	Description
`description`	`String`	The method description.

Declaring Parameters

An `<aura:method>` can optionally include parameters. Use an `<aura:attribute>` tag within an `<aura:method>` to declare a parameter for the method. For example:

```
<aura:method name="sampleMethod" action="{!c.doAction}"
  description="Sample method with parameters">
    <aura:attribute name="param1" type="String" default="parameter 1"/>
    <aura:attribute name="param2" type="Object" />
</aura:method>
```

For more information, see the **Returning a Value** section below.

Note: You don't need an `access` system attribute in the `<aura:attribute>` tag for a parameter.

Creating a Handler Action

This handler action shows how to access the arguments passed to the method.

```
({
    doAction : function(cmp, event) {
        var params = event.getParam('arguments');
        if (params) {
            var param1 = params.param1;
            // add your code here
        }
    }
})
```

Retrieve the arguments using `event.getParam('arguments')`. It returns an object if there are arguments or an empty array if there are no arguments.

Returning a Value

`aura:method` executes synchronously.

- A synchronous method finishes executing before it returns. Use the `return` statement to return a value from synchronous JavaScript code. See Return Result for Synchronous Code.
- An asynchronous method may continue to execute after it returns. Use a callback to return a value from asynchronous JavaScript code. See Return Result for Asynchronous Code.

SEE ALSO:

Calling Component Methods

Component Events

aura:set

Use `<aura:set>` in markup to set the value of an attribute inherited from a component or event.

IN THIS SECTION:

Setting Attributes Inherited from a Super Component

Use `<aura:set>` in the markup of a sub component to set the value of an inherited attribute.

Let's look at an example. Here is the `c:setTagSuper` component.

```
<!--c:setTagSuper-->
<aura:component extensible="true">
    <aura:attribute name="address1" type="String" />
    setTagSuper address1: {!v.address1}<br/>
</aura:component>
```

`c:setTagSuper` outputs:

```
setTagSuper address1:
```

The `address1` attribute doesn't output any value yet as it hasn't been set.

Here is the `c:setTagSub` component that extends `c:setTagSuper`.

```
<!--c:setTagSub-->
<aura:component extends="c:setTagSuper">
    <aura:set attribute="address1" value="808 State St" />
</aura:component>
```

`c:setTagSub` outputs:

```
setTagSuper address1: 808 State St
```

`sampleSetTagExc:setTagSub` sets a value for the `address1` attribute inherited from the super component, `c:setTagSuper`.

Warning: This usage of `<aura:set>` works for components and abstract components, but it doesn't work for interfaces. For more information, see Setting Attributes Inherited from an Interface on page 471.

If you're using a component by making a reference to it in your component, you can set the attribute value directly in the markup. For example, `c:setTagSuperRef` makes a reference to `c:setTagSuper` and sets the `address1` attribute directly without using `aura:set`.

```
<!--c:setTagSuperRef-->
<aura:component>
    <c:setTagSuper address1="1 Sesame St" />
</aura:component>
```

`c:setTagSuperRef` outputs:

```
setTagSuper address1: 1 Sesame St
```

SEE ALSO:

Component Body

Inherited Component Attributes

Setting Attributes on a Component Reference

Setting Attributes on a Component Reference

When you include another component, such as `<lightning:button>`, in a component, we call that a component reference to `<lightning:button>`. You can use `<aura:set>` to set an attribute on the component reference. For example, if your component includes a reference to `<lightning:button>`:

```
<lightning:button label="Save">
    <aura:set attribute="variant" value="brand"/>
</lightning:button>
```

This is equivalent to:

```
<lightning:button label="Save" variant="brand" />
```

The latter syntax without `aura:set` makes more sense in this simple example. You can also use this simpler syntax in component references to set values for attributes that are inherited from parent components.

`aura:set` is more useful when you want to set markup as the attribute value. For example, this sample specifies the markup for the `else` attribute in the `aura:if` tag.

```
<aura:component>
    <aura:attribute name="display" type="Boolean" default="true"/>
    <aura:if isTrue="{!v.display}">
        Show this if condition is true
        <aura:set attribute="else">
           <lightning:button label="Save" onclick="{!c.saveRecord}" />
        </aura:set>
    </aura:if>
</aura:component>
```

SEE ALSO:

Setting Attributes Inherited from a Super Component

Setting Attributes Inherited from an Interface

To set the value of an attribute inherited from an interface, redefine the attribute in the component and set its default value. Let's look at an example with the `c:myIntf` interface.

```
<!--c:myIntf-->
<aura:interface>
    <aura:attribute name="myBoolean" type="Boolean" default="true" />
</aura:interface>
```

This component implements the interface and sets `myBoolean` to `false`.

```
<!--c:myIntfImpl-->
<aura:component implements="c:myIntf">
    <aura:attribute name="myBoolean" type="Boolean" default="false" />

    <p>myBoolean: {!v.myBoolean}</p>
</aura:component>
```

JavaScript API

The JavaScript API lists the publicly accessible methods for each object that you can use in JavaScript code, such as a controller or helper. The `$A` namespace is the entry point for using the framework in JavaScript code.

IN THIS SECTION:

$A namespace

The `$A` namespace is the entry point for using the framework in JavaScript code.

Methods

IN THIS SECTION:

enqueueAction()

Queue a call to an Apex action . The framework queues up actions before sending them to the server. This mechanism is largely transparent to you when you're writing code but it enables the framework to minimize network traffic by batching multiple actions into one request (XHR).

error()

Deprecated. For a serious error that has no recovery path, throw a standard JavaScript error instead by using `throw new Error(msg)`.

get()

Returns a value from the specified global value provider using property syntax.

getCallback()

Use `$A.getCallback()` to wrap any code that modifies a component outside the normal rerendering lifecycle, such as in a `setTimeout()` call. The `$A.getCallback()` call ensures that the framework rerenders the modified component and processes any enqueued actions.

getComponent()

Gets an instance of a component from either a global ID or a DOM element that was created by a rendered component.

getReference()

Returns a live reference to the global value requested using property syntax.

getRoot()

Gets the root component or application. For example, `$A.getRoot().get("v.attrName")` returns the value of the `attrName` attribute from the root component.

getToken()

Returns an application configuration token referenced by name. A tokens file is configured with the `tokens` attribute in the `<aura:application>` tag.

log()

Deprecated. Logs to the browser's JavaScript console, if it is available. This method doesn't log in production or debug modes so it's only useful for internal usage by the framework.

reportError()

Report an error to the server after handling it. Note that the method should be used only if the try-catch mechanism of error handling is not desired or not functional, such as in nested promises.

run()

Deprecated. Use `getCallback()` instead.

set()

Sets a value on the specified global value provider using property syntax.

warning()

Deprecated. Logs a warning to the browser's JavaScript console, if it is available.

`createComponent()`

Create a component from a type and a set of attributes. This method accepts the name of a type of component, a map of attributes, and a callback to notify the caller.

Signature

`createComponent(String type, Object attributes, function callback)`

Parameters

`type`
: Type: `String`

  The type of component to create. For example, `"lightning:button"`.

`attributes`
: Type: `Object`

  A map of attributes to send to the component. These attributes take the same form as in the markup, including events `{"press":component.getReference("c.handlePress")}`, and id `{"aura:id":"myComponentId"}`.

`callback(cmp, status, errorMessage)`
: Type: `function`

  The callback to invoke after the component is created. The callback has three parameters.

  1. `cmp`—The component that was created. This parameter enables you to do something with the new component, such as add it to the body of the component that creates it. If there's an error, `cmp` is `null`.
  2. `status`—The status of the call. The possible values are `SUCCESS`, `INCOMPLETE`, or `ERROR`. Always check that the status is `SUCCESS` before you try to use the component.
  3. `errorMessage`—The error message if the status is `ERROR`.

SEE ALSO:

Dynamically Creating Components

`createComponents()`

Create an array of components from a list of types and attributes. This method accepts a list of component names and attribute maps, and a callback to notify the caller.

Signature

`createComponents(Array components, function callback)`

Parameters

`components`
: Type: `Array`

  The list of components to create. For example, `["lightning:button", {"onclick":component.getReference("c.handlePress")}]`

`callback(components, status, errorMessage)`
: Type: `function`

  The callback to invoke after the components are created. The callback has three parameters.

  1. `components`—The components that were created. This parameter enables you to do something with the new components, such as add them to the body of the component that created them. If there's an error, `components` is `null`.

2. `status`—The status of the call. The possible values are `SUCCESS`, `INCOMPLETE`, or `ERROR`. Always check that the status is `SUCCESS` before you try to use the components.
3. `errorMessage`—The error message if the status is `ERROR`.

SEE ALSO:

Dynamically Creating Components

`enqueueAction()`

Queue a call to an Apex action . The framework queues up actions before sending them to the server. This mechanism is largely transparent to you when you're writing code but it enables the framework to minimize network traffic by batching multiple actions into one request (XHR).

The batching of actions is also known as boxcar'ing, similar to a train that couples boxcars together.

The framework uses a stack to keep track of the actions to send to the server. When the browser finishes processing events and JavaScript on the client, the enqueued actions on the stack are sent to the server in a batch.

Signature

```
enqueueAction (Action action, Boolean background)
```

Parameters

`action`
Type: `Action`

The action to enqueue.

`background`
Type: `Boolean`

Deprecated. Do not use.

SEE ALSO:

Queueing of Server-Side Actions

Calling a Server-Side Action

`error()`

Deprecated. For a serious error that has no recovery path, throw a standard JavaScript error instead by using `throw new Error(msg)`.

Signature

```
error (String msg, Error e)
```

Parameters

`msg`
Type: `String`

The error message to display to the user.

`e`

Type: `Error`

The error message to display to the user.

`get()`

Returns a value from the specified global value provider using property syntax.

Signature

```
get (String key, function callback)
```

Parameters

`key`

Type: `String`

The data key to look up. For example, `"$Label.c.labelName"` for a custom label.

`callback`

Type: `function`

The method to call with the result if a server trip occurs.

Returns

Type: `String`

The requested value.

SEE ALSO:

set()

`getCallback()`

Use `$A.getCallback()` to wrap any code that modifies a component outside the normal rerendering lifecycle, such as in a `setTimeout()` call. The `$A.getCallback()` call ensures that the framework rerenders the modified component and processes any enqueued actions.

Don't use `$A.getCallback()` if your code is executed as part of the framework's call stack. For example, your code is handling an event or in the callback for an Apex controller action.

Signature

```
getCallback (function callback)
```

Parameters

`callback`

Type: `function`

The method to call after establishing an Aura context.

Sample Code

```
window.setTimeout(
    $A.getCallback(function() {
        cmp.set("v.visible", true);
    }), 5000
);
```

SEE ALSO:

Modifying Components Outside the Framework Lifecycle

`getComponent()`

Gets an instance of a component from either a global ID or a DOM element that was created by a rendered component.

Signature

`getComponent (Object identifier)`

Parameters

`identifier`

Type: `Object`

A globalId or an element.

`getReference()`

Returns a live reference to the global value requested using property syntax.

Signature

`getReference (String key)`

Parameters

`key`

Type: `String`

The data key for which to return a reference.

Returns

Type: `PropertyReferenceValue`

The reference to the global value requested.

`getRoot()`

Gets the root component or application. For example, `$A.getRoot().get("v.attrName")` returns the value of the `attrName` attribute from the root component.

Signature

`getRoot()`

`getToken()`

Returns an application configuration token referenced by name. A tokens file is configured with the `tokens` attribute in the `<aura:application>` tag.

Signature

`getToken (String token)`

Parameters

`token`
Type: `String`

The name of the application configuration token to retrieve.

Returns

Type: `String`
application configuration token.

`log()`

Deprecated. Logs to the browser's JavaScript console, if it is available. This method doesn't log in production or debug modes so it's only useful for internal usage by the framework.

Signature

`log (Object value, Object error)`

Parameters

`value`
Type: `Object`

The object to log.

`error`
Type: `Object`

The error message to log in the stack trace.

Returns

Type: `String`

The requested value.

`reportError()`

Report an error to the server after handling it. Note that the method should be used only if the try-catch mechanism of error handling is not desired or not functional, such as in nested promises.

Signature

```
reportError (String message, Error error)
```

Parameters

`message`

Type: `String`

The error message.

`error`

Type: `Error`

An error object to be included in handling and reporting.

`run()`

Deprecated. Use `getCallback()` instead.

Signature

```
run (function func, String name)
```

Parameters

`func`

Type: `function`

The function to run.

`name`

Type: `String`

An optional name for the stack.

`set()`

Sets a value on the specified global value provider using property syntax.

Signature

```
set (String key, Object value)
```

Parameters

`key`
: Type: `String`

  The data key to change on the global value provider.

`value`
: Type: `Object`

  The value to set for the key. If the global value provider doesn’t implement `set()`, this method throws an exception.

`warning()`

Deprecated. Logs a warning to the browser's JavaScript console, if it is available.

Signature

`warning (String w, Error e)`

Parameters

`w`
: Type: `String`

  The message to log.

`error`
: Type: `Object`

  The error message to log in the stack trace.

Returns

Type: `String`
: The requested value.

Action

`Action` contains methods to work with JavaScript actions that you can use to communicate with Apex classes.

Methods

IN THIS SECTION:

getError()
: Returns an array of error objects for server-side actions only. Each error object has a message field. In any mode except `PROD` mode, each object also has a stack field, which is a list describing the execution stack when the error occurred.

getName()
: Returns the name of an action.

getParam()
: Returns an action parameter value for a parameter name.

getParams()
: Returns the collection of parameters for an action.

getReturnValue()
: Gets the return value of an Apex action. An Apex action can return any object containing serializable JSON data.

getState()
: Returns the current state of an action. Check the state of the action in the callback after an Apex action completes.

isBackground()
: Returns `true` if the action is enqueued in the background, `false` if it's enqueued in the foreground.

setAbortable()
: Sets an action as abortable. If the component is not valid, abortable actions are not sent to the server. A component is automatically destroyed and marked invalid by the framework when it is unrendered. Actions not marked abortable are always sent to the server regardless of the validity of the component.

setBackground()
: Sets the action to run as a background action. This cannot be unset. Background actions are usually long running and lower priority actions. A background action is useful when you want your app to remain responsive to a user while it executes a low priority, long-running action. A rough guideline is to use a background action if it takes more than five seconds for the response to return from the server.

setCallback()
: Sets the callback function that is executed after an Apex action returns.

setParam()
: Sets a single parameter for an action. Use parameters to pass data to an Apex action.

setParams()
: Sets parameters for an action. Use parameters to pass data to an Apex action.

setStorable()
: Marks an Apex action as storable to have its response stored in the framework's client-side cache . Enhance your component's performance by marking actions as storable (cacheable) to quickly show cached data from client-side storage without waiting for a server trip. If the cached data is stale, the framework retrieves the latest data from the server. Caching is especially beneficial for users on high latency, slow, or unreliable connections such as 3G networks.

`getError()`

Returns an array of error objects for server-side actions only. Each error object has a message field. In any mode except `PROD` mode, each object also has a stack field, which is a list describing the execution stack when the error occurred.

Signature

`getError()`

Returns

Type: `Object[]`
: An array of error objects. Each error object has a message field.

`getName()`

Returns the name of an action.

Signature

`getName()`

Returns

Type: `String`
The action name.

`getParam()`

Returns an action parameter value for a parameter name.

Signature

`getParam (String name)`

Parameters

`name`
Type: `String`

The parameter name.

Returns

Type: `Object`
The parameter value.

`getParams()`

Returns the collection of parameters for an action.

Signature

`getParams`

Returns

Type: `Object`
The key-value pairs for the action parameters.

`getReturnValue()`

Gets the return value of an Apex action. An Apex action can return any object containing serializable JSON data.

Signature

`getReturnValue()`

Returns

Type: `Object`

The return value of an Apex action.

SEE ALSO:

Calling a Server-Side Action

`getState()`

Returns the current state of an action. Check the state of the action in the callback after an Apex action completes.

Signature

`getState()`

Returns

Type: `String`

The action state.

SEE ALSO:

Action States

`isBackground()`

Returns `true` if the action is enqueued in the background, `false` if it's enqueued in the foreground.

Signature

`isBackground()`

Returns

Type: `Boolean`

Returns `true` if the action is enqueued in the background.

`setAbortable()`

Sets an action as abortable. If the component is not valid, abortable actions are not sent to the server. A component is automatically destroyed and marked invalid by the framework when it is unrendered. Actions not marked abortable are always sent to the server regardless of the validity of the component.

For example, a save or edit action should not be set as abortable to ensure that it's always sent to the server even if the component is deleted. Setting an action as abortable can't be undone.

Signature

`setAbortable()`

SEE ALSO:

Abortable Actions

`setBackground()`

Sets the action to run as a background action. This cannot be unset. Background actions are usually long running and lower priority actions. A background action is useful when you want your app to remain responsive to a user while it executes a low priority, long-running action. A rough guideline is to use a background action if it takes more than five seconds for the response to return from the server.

Signature

`setBackground()`

`setCallback()`

Sets the callback function that is executed after an Apex action returns.

Signature

`setCallback (Object scope, function callback, String name)`

Parameters

`scope`
: Type: `Object`

  The scope in which the function is executed. Always set this parameter to the keyword `this`.

`callback`
: Type: `function`

  The callback to invoke after the Apex action returns.

`name`
: Type: `String`

  Defaults to "ALL" which registers callbacks for the "SUCCESS", "ERROR", and "INCOMPLETE" states.

SEE ALSO:

Calling a Server-Side Action

Action States

`setParam()`

Sets a single parameter for an action. Use parameters to pass data to an Apex action.

Signature

`setParam (String key, Object value)`

Parameters

`key`
: Type: `String`

  The parameter name.

`value`
: Type: `Object`

  The parameter value.

SEE ALSO:

Calling a Server-Side Action

`setParams()`

Sets parameters for an action. Use parameters to pass data to an Apex action.

Signature

`setParams (Object config)`

Parameters

`config`
: Type: `Object`

  The key-value pairs for action parameters. For example `{ "record": ` ***`id,`*** ` "name": ` ***`name`***`}.`

SEE ALSO:

Calling a Server-Side Action

`setStorable()`

Marks an Apex action as storable to have its response stored in the framework's client-side cache . Enhance your component's performance by marking actions as storable (cacheable) to quickly show cached data from client-side storage without waiting for a server trip. If the cached data is stale, the framework retrieves the latest data from the server. Caching is especially beneficial for users on high latency, slow, or unreliable connections such as 3G networks.

Note: Client-side storage is automatically configured in Lightning Experience and the Salesforce mobile app. A component shouldn't assume a cache duration because it may change as we optimize the platform.

Signature

`setStorable (Object config)`

Parameters

`config`

Type: `Object`

An optional configuration map of key-value pairs representing the storage options and values to set. You can only set the `ignoreExisting` property. Set `ignoreExisting` to `true` to bypass the cache. The default value is `false`.

This property is useful when you know that any cached data is invalid, such as after a record modification. This property should be used rarely because it explicitly defeats caching.

SEE ALSO:

Storable Actions

AuraLocalizationService

`AuraLocalizationService` provides methods for formatting and localizing dates. Use `$A.localizationService` to use the methods in `AuraLocalizationService`.

Methods

IN THIS SECTION:

formatCurrency()
Returns a currency number based on the default currency format.

formatDate()
Returns a formatted date.

formatDateTime()
Returns a formatted date time.

formatDateTimeUTC()
Returns a formatted date time in UTC.

formatDateUTC()
Returns a formatted date in UTC.

formatNumber()
Returns a formatted number with the default number format.

formatPercent()
Returns a formatted percentage number based on the default percentage format.

formatTime()
Returns a formatted time.

formatTimeUTC()
Returns a formatted time in UTC.

getDateStringBasedOnTimezone
Gets a date string based on a time zone.

getDaysInDuration()
Returns the number of days in a duration.

getDefaultCurrencyFormat()
Returns the default currency format.

getDefaultNumberFormat()
Returns the default `NumberFormat` object.

getDefaultPercentFormat()
Returns the default percentage format.

getHoursInDuration()
Returns a length of time in hours.

getLocalizedDateTimeLabels()
Deprecated. Do not use. Returns date time labels, such as month name, weekday name.

getMillisecondsInDuration()
Returns the number of milliseconds in a duration.

getMinutesInDuration()
Returns the number of minutes in a duration.

getMonthsInDuration()
Returns the number of months in a duration.

getNumberFormat()
Returns a `NumberFormat` object.

getSecondsInDuration()
Returns the number of seconds in a duration.

getToday
Gets today's date based on a time zone.

getYearsInDuration()
Returns the number of years in a duration.

isAfter()
Checks if `date1` is after `date2`.

isBefore()
Checks if `date1` is before `date2`.

isBetween()
Checks if `date` is between `fromDate` and `toDate`, where the match is inclusive.

isPeriodTimeView()
Deprecated. Do not use. Checks if a datetime pattern string uses a 24-hour or 12-hour time view.

isSame()
Checks if `date1` is the same as `date2`.

parseDateTime()
Parses a string and returns a JavaScript Date.

parseDateTimeISO8601()
Parses a date time string in an ISO-8601 format and returns a JavaScript Date.

parseDateTimeUTC()
Parses a string and returns a JavaScript Date.

startOf()
Returns a date that is the start of a unit of time for the given date.

toISOString()
Deprecated. Use `Date.toISOString()` instead.

translateFromLocalizedDigits()
Translate the localized digit string to a string with Arabic digits, if there is any.

translateFromOtherCalendar()
Translates the input date from another calendar system (for example, the Buddhist calendar) to the Gregorian calendar based on the locale.

translateToLocalizedDigits()
Translate the input string to a string with localized digits, if there is any.

translateToOtherCalendar()
Translates the input date to a date in another calendar system (for example, the Buddhist calendar) based on the locale.

SEE ALSO:

Formatting Dates in JavaScript

UTCToWallTime()

Converts a datetime from UTC to a specified timezone.

Signature

`UTCToWallTime (Date date, String timezone, function callback)`

Parameters

`date`
: Type: `Date`

  A JavaScript `Date` object.

`timezone`
: Type: `String`

  A time zone ID based on the class, for example, `"America/Los_Angeles"`.

`callback`
: Type: `function`

  A function to call after the conversion is done. Access the converted value in the first parameter of the callback.

Sample Code

```
var format = $A.get("$Locale.timeFormat");
format = format.replace(":ss", "");
var langLocale = $A.get("$Locale.langLocale");
var timezone = $A.get("$Locale.timezone");
var date = new Date();
$A.localizationService.UTCToWallTime(date, timezone, function(walltime) {
    // Returns the local time without the seconds, for example, 9:00 PM
   displayValue = $A.localizationService.formatDateTimeUTC(walltime, format, langLocale);
})
```

`WallTimeToUTC`

Converts a datetime from a specified timezone to UTC.

Signature

`WallTimeToUTC (Date date, string timezone, function callback)`

Parameters

`date`
: Type: `Date`

  A JavaScript `Date` object.

`timezone`
: Type: `String`

A time zone ID based on the class, for example, `"America/Los_Angeles"`.

`callback`

Type: `function`

A function to call after the conversion is done. Access the converted value in the first parameter of the callback.

`displayDuration()`

Displays a length of time.

Signature

`displayDuration (Duration duration, boolean withSuffix)`

Parameters

`duration`

Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

`withSuffix`

Type: `boolean`

If `true`, returns value with a suffix matching the unit of the *`duration`* parameter.

Returns

Type: `String`

The length of time.

Sample Code

```
var dur = $A.localizationService.duration(1, 'day');
// Returns "a day"
var length = $A.localizationService.displayDuration(dur);
```

SEE ALSO:

duration()

`displayDurationInDays()`

Displays a length of time in days.

Signature

`displayDurationInDays (Duration duration)`

Parameters

`duration`

Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

Returns

Type: `number`

The length of time in days.

Sample Code

```
var dur = $A.localizationService.duration(24, 'hour');
// Returns 1
var length = $A.localizationService.displayDurationInDays(dur);
```

SEE ALSO:

duration()

`displayDurationInHours()`

Displays a length of time in hours.

Signature

`displayDurationInHours (Duration duration)`

Parameters

`duration`

Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

Returns

Type: `number`

The length of time in hours.

Sample Code

```
var dur = $A.localizationService.duration(2, 'day');
// Returns 48
var length = $A.localizationService.displayDurationInHours(dur);
```

SEE ALSO:

duration()

`displayDurationInMilliseconds()`

Displays a length of time in milliseconds.

Signature

`displayDurationInMilliseconds (Duration duration)`

Parameters

`duration`

Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

Returns

Type: `number`

The length of time in milliseconds.

Sample Code

```
var dur = $A.localizationService.duration(1, 'hour');
// Returns 3600000
var length = $A.localizationService.displayDurationInMilliseconds(dur);
```

SEE ALSO:

duration()

`displayDurationInMinutes()`

Displays a length of time in minutes.

Signature

`displayDurationInMinutes (Duration duration)`

Parameters

`duration`

Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

Returns

Type: `number`

The length of time in minutes.

Sample Code

```
var dur = $A.localizationService.duration(1, 'hour');
// Returns 60
var length = $A.localizationService.displayDurationInMinutes(dur);
```

SEE ALSO:

duration()

`displayDurationInMonths()`

Displays a length of time in months.

Signature

`displayDurationInMonths (Duration duration)`

Parameters

`duration`

Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

Returns

Type: `number`

The length of time in months.

Sample Code

```
var dur = $A.localizationService.duration(60, 'day');
// Returns 1.971293
var length = $A.localizationService.displayDurationInMonths(dur);
```

SEE ALSO:

duration()

`displayDurationInSeconds()`

Displays a length of time in seconds.

Signature

`displayDurationInSeconds (Duration duration)`

Parameters

duration
: Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

Returns

Type: `number`
The length of time in seconds.

Sample Code

```
var dur = $A.localizationService.duration(60, 'minutes');
// Returns 3600
var length = $A.localizationService.displayDurationInSeconds(dur);
```

SEE ALSO:
duration()

`duration()`

Returns an object representing a length of time.

Signature

`duration (number num, String unit)`

Parameters

num
Type: `number`

The length of time in a given unit.

unit
Type: `String`

A datetime unit. The default is 'milliseconds'. The options are 'years, 'months', 'weeks', 'days', 'hour', 'minutes', 'seconds', 'milliseconds'.

Returns

Type: `Object`
A duration object.

Sample Code

```
var dur = $A.localizationService.duration(2, 'days');
```

`endOf()`

Returns a date that is the end of a unit of time for the given date.

Signature

`endOf(string | number | Date date, string unit)`

Parameters

`date`
    Type: `string | number | Date`

    A datetime string in ISO8601 format, or a timestamp in milliseconds, or a `Date` object.

`unit`
    Type: `string`

    A datetime unit. Options are 'year', 'month', 'week', 'day', 'hour', 'minute', or 'second'.

Returns

Type: `Date`
    A JavaScript `Date` object. If a unit is not provided, returns a parsed date.

Sample Code

```
var date = new Date();
// Returns the time at the end of the day
// in the format "Fri Oct 09 2015 23:59:59 GMT-0700 (PDT)"
var day = $A.localizationService.endOf(date, 'day');
```

`formatCurrency()`

Returns a currency number based on the default currency format.

Signature

`formatCurrency (number number)`

Parameters

`number`
    Type: `number`

    The currency number to format.

Returns

Type: `number`
    The formatted currency.

Sample Code

```
var curr = 123.45;
// Returns $123.45
$A.localizationService.formatCurrency(curr);
```

`formatDate()`

Returns a formatted date.

Signature

`formatDate (string | number | Date date, string formatString, string locale)`

Parameters

`date`
: Type: `string | number | Date`

  A datetime string in ISO8601 format , or a timestamp in milliseconds, or a `Date` object. If you provide a String value, use ISO 8601 format to avoid parsing warnings. If no timezone is specified, defaults to the browser timezone offset.

`formatString`
: Type: `string`

  Optional. A string containing tokens to format the given date. For example, `"yyyy-MM-dd"` formats 15th January, 2017 as "2017-01-15". The default format string comes from the `$Locale` value provider. For details on available tokens, see Formatting Dates in JavaScript.

`locale`
: Type: `string`

  Optional. A locale to format the given date. The default value is `$Locale.langLocale`. We strongly recommended that you use the locale value from `$Locale`. We fall back to the value in `$Locale.langLocale` if you use an unavailable locale.

Returns

Type: `string`
: A formatted and localized date string.

Sample Code

```
var date = new Date();
// Returns date in the format "Oct 9, 2015"
$A.localizationService.formatDate(date);
```

`formatDateTime()`

Returns a formatted date time.

Signature

`formatDateTime (string | number | Date date, string formatString, string locale)`

Parameters

`date`
: Type: `string | number | Date`

A datetime string in ISO8601 format , or a timestamp in milliseconds, or a `Date` object. If you provide a String value, use ISO 8601 format to avoid parsing warnings. If no timezone is specified, defaults to the browser timezone offset.

`formatString`
: Type: `string`

Optional. A string containing tokens to format the given date. For example, `"yyyy-MM-dd"` formats 15th January, 2017 as "2017-01-15". The default format string comes from the `$Locale` value provider. For details on available tokens, see Formatting Dates in JavaScript.

`locale`
: Type: `string`

Optional. A locale to format the given date. The default value is `$Locale.langLocale`. We strongly recommended that you use the locale value from `$Locale`. We fall back to the value in `$Locale.langLocale` if you use an unavailable locale.

Returns

Type: `string`

A formatted and localized date time string.

Sample Code

```
var date = new Date();
// Returns datetime in the format "Oct 9, 2015 9:00:00 AM"
$A.localizationService.formatDateTime(date);
```

`formatDateTimeUTC()`

Returns a formatted date time in UTC.

Signature

`formatDateTimeUTC (string | number | Date date, string formatString, string locale)`

Parameters

`date`
: Type: `string | number | Date`

A datetime string in ISO8601 format , or a timestamp in milliseconds, or a `Date` object. If you provide a String value, use ISO 8601 format to avoid parsing warnings. If no timezone is specified, defaults to the browser timezone offset.

`formatString`
: Type: `string`

Optional. A string containing tokens to format the given date. For example, `"yyyy-MM-dd"` formats 15th January, 2017 as "2017-01-15". The default format string comes from the `$Locale` value provider. For details on available tokens, see Formatting Dates in JavaScript.

`locale`

Type: `string`

Optional. A locale to format the given date. The default value is `$Locale.langLocale`. We strongly recommended that you use the locale value from `$Locale`. We fall back to the value in `$Locale.langLocale` if you use an unavailable locale.

Returns

Type: `string`

A formatted and localized date time string.

Sample Code

```
var date = new Date();
// Returns datetime in UTC in the format "Oct 9, 2015 4:00:00 PM"
$A.localizationService.formatDateTimeUTC(date);
```

`formatDateUTC()`

Returns a formatted date in UTC.

Signature

`formatDateUTC (string | number | Date date, string formatString, string locale)`

Parameters

`date`

Type: `string | number | Date`

A datetime string in ISO8601 format , or a timestamp in milliseconds, or a `Date` object. If you provide a String value, use ISO 8601 format to avoid parsing warnings. If no timezone is specified, defaults to the browser timezone offset.

`formatString`

Type: `string`

Optional. A string containing tokens to format the given date. For example, `"yyyy-MM-dd"` formats 15th January, 2017 as "2017-01-15". The default format string comes from the `$Locale` value provider. For details on available tokens, see Formatting Dates in JavaScript.

`locale`

Type: `string`

Optional. A locale to format the given date. The default value is `$Locale.langLocale`. We strongly recommended that you use the locale value from `$Locale`. We fall back to the value in `$Locale.langLocale` if you use an unavailable locale.

Returns

Type: `string`

A formatted and localized date string.

Sample Code

```
var date = new Date();
// Returns date in UTC in the format "Oct 9, 2015"
$A.localizationService.formatDateUTC(date);
```

`formatNumber()`

Returns a formatted number with the default number format.

Signature

`formatNumber (number number)`

Parameters

`number`
Type: `number`

The number to format.

Returns

Type: `number`
The formatted number.

Sample Code

```
var num = 10000;
// Returns 10,000
var formatted = $A.localizationService.formatNumber(num);
```

`formatPercent()`

Returns a formatted percentage number based on the default percentage format.

Signature

`formatPercent (number number)`

Parameters

`number`
Type: `number`

The number to format.

Returns

Type: `number`
The formatted percentage.

Sample Code

```
var num = 0.54;
// Returns 54%
var formatted = $A.localizationService.formatPercent(num);
```

formatTime()

Returns a formatted time.

Signature

`formatTime (string | number | Date date, string formatString, string locale)`

Parameters

date
: Type: `string | number | Date`

  A datetime string in ISO8601 format , or a timestamp in milliseconds, or a `Date` object. If you provide a String value, use ISO 8601 format to avoid parsing warnings. If no timezone is specified, defaults to the browser timezone offset.

formatString
: Type: `string`

  Optional. A string containing tokens to format the given date. For example, `"yyyy-MM-dd"` formats 15th January, 2017 as "2017-01-15". The default format string comes from the `$Locale` value provider. For details on available tokens, see Formatting Dates in JavaScript.

locale
: Type: `string`

  Optional. A locale to format the given date. The default value is `$Locale.langLocale`. We strongly recommended that you use the locale value from `$Locale`. We fall back to the value in `$Locale.langLocale` if you use an unavailable locale.

Returns

Type: `string`
: A formatted and localized time string.

Sample Code

```
var date = new Date();
// Returns a date in the format "9:00:00 AM"
var now = $A.localizationService.formatTime(date);
```

formatTimeUTC()

Returns a formatted time in UTC.

Signature

`formatTime (string | number | Date date, string formatString, string locale)`

Parameters

`date`
: Type: `string | number | Date`

  A datetime string in ISO8601 format , or a timestamp in milliseconds, or a `Date` object. If you provide a String value, use ISO 8601 format to avoid parsing warnings. If no timezone is specified, defaults to the browser timezone offset.

`formatString`
: Type: `string`

  Optional. A string containing tokens to format the given date. For example, `"yyyy-MM-dd"` formats 15th January, 2017 as "2017-01-15". The default format string comes from the `$Locale` value provider. For details on available tokens, see Formatting Dates in JavaScript.

`locale`
: Type: `string`

  Optional. A locale to format the given date. The default value is `$Locale.langLocale`. We strongly recommended that you use the locale value from `$Locale`. We fall back to the value in `$Locale.langLocale` if you use an unavailable locale.

Returns

Type: `string`
: A formatted and localized time string.

Sample Code

```
var date = new Date();
// Returns time in UTC in the format "4:00:00 PM"
$A.localizationService.formatTimeUTC(date);
```

getDateStringBasedOnTimezone

Gets a date string based on a time zone.

Signature

`getDateStringBasedOnTimezone (string timeZone, Date date, function callback)`

Parameters

`timezone`
: Type: `String`

  A time zone ID based on the class, for example, `"America/Los_Angeles"`.

`date`
: Type: `Date`

  A JavaScript `Date` object.

callback

Type: `function`

A function to call after the date string is returned. Access the date string in the first parameter of the callback.

Sample Code

```
var timezone = $A.get("$Locale.timezone");
var date = new Date();
// Returns the date string in the format "2015-10-9"
$A.localizationService.getDateStringBasedOnTimezone(timezone, date, function(today){
    console.log(today);
});
```

`getDaysInDuration()`

Returns the number of days in a duration.

Signature

`getDaysInDuration(Duration duration)`

Parameters

duration

Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

Returns

Type: `number`

The number of days in the duration.

Sample Code

```
var dur = $A.localizationService.duration(48, 'hour');
// Returns 2, the number of days for the given duration
$A.localizationService.getDaysInDuration(dur);
```

SEE ALSO:

duration()

`getDefaultCurrencyFormat()`

Returns the default currency format.

Signature

`getDefaultCurrencyFormat()`

Returns

Type: `NumberFormat`
The currency format returned by `$Locale.currencyFormat`.

Sample Code

```
// Returns $20,000.00
$A.localizationService.getDefaultCurrencyFormat().format(20000);
```

SEE ALSO:
$Locale

`getDefaultNumberFormat()`

Returns the default `NumberFormat` object.

Signature

`getDefaultNumberFormat()`

Returns

Type: `NumberFormat`
The number format returned by `$Locale.numberFormat`.

Sample Code

```
// Returns 20,000.123
$A.localizationService.getDefaultNumberFormat().format(20000.123);
```

SEE ALSO:
$Locale

`getDefaultPercentFormat()`

Returns the default percentage format.

Signature

`getDefaultPercentFormat()`

Returns

Type: `NumberFormat`
The percentage format returned by `$Locale.percentFormat`.

Sample Code

```
// Returns 20%
$A.localizationService.getDefaultPercentFormat().format(0.20);
```

SEE ALSO:

$Locale

`getHoursInDuration()`

Returns a length of time in hours.

Signature

`getHoursInDuration(Duration duration)`

Parameters

`duration`

Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

Returns

Type: `number`

The number of hours in the duration.

Sample Code

```
var dur = $A.localizationService.duration(60, 'minute');
// Returns 1, the number of hours in the given duration
$A.localizationService.getHoursInDuration(dur);
```

SEE ALSO:

duration()

`getLocalizedDateTimeLabels()`

Deprecated. Do not use. Returns date time labels, such as month name, weekday name.

Signature

`getLocalizedDateTimeLabels()`

Returns

Type: `Object`

The localized set of labels.

`getMillisecondsInDuration()`

Returns the number of milliseconds in a duration.

Signature

`getMillisecondsInDuration(Duration duration)`

Parameters

`duration`
Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

Returns

Type: `number`
The number of milliseconds in the duration.

SEE ALSO:

duration()

`getMinutesInDuration()`

Returns the number of minutes in a duration.

Signature

`getMinutesInDuration(Duration duration)`

Parameters

`duration`
Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

Returns

Type: `number`
The number of minutes in the duration.

Sample Code

```
var dur = $A.localizationService.duration(60, 'second');
// Returns 1, the number of minutes in the given duration
$A.localizationService.getMinutesInDuration(dur);
```

SEE ALSO:

duration()

`getMonthsInDuration()`

Returns the number of months in a duration.

Signature

`getMonthsInDuration(Duration duration)`

Parameters

`duration`

Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

Returns

Type: `number`

The number of months in the duration.

Sample Code

```
var dur = $A.localizationService.duration(70, 'day');
// Returns 2, the number of months in the given duration
$A.localizationService.getMonthsInDuration(dur);
```

SEE ALSO:

duration()

`getNumberFormat()`

Returns a `NumberFormat` object.

Signature

`getNumberFormat(string format, string symbols)`

Parameters

`format`

Type: `string`

The number format. For example, `format=".00"` displays the number followed by two decimal places.

`symbols`

Type: `string`

An optional map of localized symbols. Otherwise, the current locale's symbols are used.

Returns

Type: `NumberFormat`

The number format returned by `$Locale.numberFormat`.

Sample Code

```
var f = $A.get("$Locale.numberFormat");
var num = 10000
var nf = $A.localizationService.getNumberFormat(f);
var formatted = nf.format(num);
// Returns 10,000
var formatted = $A.localizationService.formatNumber(num);
```

`getSecondsInDuration()`

Returns the number of seconds in a duration.

Signature

`getSecondsInDuration(Duration duration)`

Parameters

`duration`

Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

Returns

Type: `number`

The number of seconds in the duration.

Sample Code

```
var dur = $A.localizationService.duration(3000, 'millisecond');
// Returns 3
$A.localizationService.getSecondsInDuration(dur);
```

SEE ALSO:

duration()

getToday

Gets today's date based on a time zone.

Signature

`getToday(string timezone, function callback)`

Parameters

`timezone`

Type: `String`

A time zone ID based on the class, for example, `"America/Los_Angeles"`.

`callback`

Type: `function`

A function to call after the date is returned. Access the date in the first parameter of the callback.

Sample Code

```
var timezone = $A.get("$Locale.timezone");
// Returns the date string in the format "2015-11-25"
$A.localizationService.getToday(timezone, function(today){
    console.log(today);
});
```

getYearsInDuration()

Returns the number of years in a duration.

Signature

`getYearsInDuration(Duration duration)`

Parameters

`duration`

Type: `Duration`

The duration object returned by `$A.localizationService.duration`.

Returns

Type: `number`

The number of years in the duration.

Sample Code

```
var dur = $A.localizationService.duration(24, 'month');
// Returns 2
$A.localizationService.getYearsInDuration(dur);
```

SEE ALSO:

duration()

`isAfter()`

Checks if `date1` is after `date2`.

Signature

`isAfter(string | number | Date date1, string | number | Date date2, string unit)`

Parameters

`date1`

Type: `string | number | Date`

A datetime string in ISO8601 format, or a timestamp in milliseconds, or a `Date` object.

`date2`

Type: `string | number | Date`

A datetime string in ISO8601 format, or a timestamp in milliseconds, or a `Date` object.

`unit`

Type: `string`

A datetime unit. Options are 'year', 'month', 'week', 'day', 'hour', 'minute', 'second', or 'millisecond'.

Returns

Type: `boolean`

Returns `true` if `date1` is after `date2`, or `false` otherwise.

Sample Code

```
var date = new Date();
var day = $A.localizationService.endOf(date, 'day');
// Returns false, since date is before day
$A.localizationService.isAfter(date, day);
```

`isBefore()`

Checks if `date1` is before `date2`.

Signature

`isBefore(string | number | Date date1, string | number | Date date2, string unit)`

Parameters

`date1`
Type: `string | number | Date`

A datetime string in ISO8601 format, or a timestamp in milliseconds, or a `Date` object.

`date2`
Type: `string | number | Date`

A datetime string in ISO8601 format, or a timestamp in milliseconds, or a `Date` object.

`unit`
Type: `string`

A datetime unit. Options are 'year', 'month', 'week', 'day', 'hour', 'minute', 'second', or 'millisecond'.

Returns

Type: `boolean`
Returns `true` if `date1` is before `date2`, or `false` otherwise.

Sample Code

```
var date = new Date();
var day = $A.localizationService.endOf(date, 'day');
// Returns true, since date is before day
$A.localizationService.isBefore(date, day);
```

`isBetween()`

Checks if `date` is between `fromDate` and `toDate`, where the match is inclusive.

Signature

`isBetween(string | number | Date date, string | number | Date fromDate, string | number | Date toDate, string unit)`

Parameters

`date`
Type: `string | number | Date`

A datetime string in ISO8601 format, or a timestamp in milliseconds, or a `Date` object.

fromDate
Type: `string | number | Date`

A datetime string in ISO8601 format, or a timestamp in milliseconds, or a `Date` object.

toDate
Type: `string | number | Date`

A datetime string in ISO8601 format, or a timestamp in milliseconds, or a `Date` object.

unit
Type: `string`

A datetime unit. Options are 'year', 'month', 'week', 'day', 'hour', 'minute', 'second', or 'millisecond'.

Returns

Type: `boolean`
Returns `true` if `date` is between `fromDate` and `toDate`, or `false` otherwise.

Sample Code

```
// Returns true
$A.localizationService.isBetween("2017-03-07","March 7, 2017", "12/1/2017");
// Returns false
$A.localizationService.isBetween("2017-03-07 12:00", "March 7, 2017 15:00", "12/1/2017");
// Returns true because the unit is "day"
$A.localizationService.isBetween("2017-03-07 12:00", "March 7, 2017 15:00", "12/1/2017",
"day");
```

`isPeriodTimeView()`

Deprecated. Do not use. Checks if a datetime pattern string uses a 24-hour or 12-hour time view.

Signature

`isPeriodTimeView(string pattern)`

Parameters

pattern
Type: `string`

A datetime pattern.

Returns

Type: `boolean`
Returns `true` if the pattern uses a 12-hour period time view.

`isSame()`

Checks if `date1` is the same as `date2`.

Signature

`isSame(string | number | Date date1, string | number | Date date2, string unit)`

Parameters

`date1`
Type: `string | number | Date`

A datetime string in ISO8601 format, or a timestamp in milliseconds, or a `Date` object.

`date2`
Type: `string | number | Date`

A datetime string in ISO8601 format, or a timestamp in milliseconds, or a `Date` object.

`unit`
Type: `string`

A datetime unit. Options are 'year', 'month', 'week', 'day', 'hour', 'minute', 'second', or 'millisecond'.

Returns

Type: `boolean`
Returns `true` if `date1` is the same as `date2`, or `false` otherwise.

Sample Code

```
var date = new Date();
var day = $A.localizationService.endOf(date, 'day');
// Returns false
$A.localizationService.isSame(date, day, 'hour');
// Returns true
$A.localizationService.isSame(date, day, 'day');
```

`parseDateTime()`

Parses a string and returns a JavaScript Date.

Signature

`parseDateTime(string dateTimeString, string parseFormat, string | boolean locale, boolean strictParsing)`

Parameters

`dateTimeString`
Type: `string`

A datetime string.

`parseFormat`
Type: `string`

An optional Java format string used to parse the datetime. The default is from the `$Locale` global value provider.

`locale`
Type: `string | boolean`

This parameter is deprecated.

`strictParsing`
Type: `string`

Set this optional parameter to `true` to turn off forgiving parsing and use strict validation.

Returns

Type: `Date`
Returns a JavaScript `Date` object, or `null` if `dateTimeString` is invalid.

`parseDateTimeISO8601()`

Parses a date time string in an ISO-8601 format and returns a JavaScript Date.

Signature

```
parseDateTimeISO8601(string dateTimeString)
```

Parameters

`dateTimeString`
Type: `string`

A datetime string in ISO8601 format.

Returns

Type: `Date`
Returns a JavaScript `Date` object, or `null` if `dateTimeString` is invalid.

`parseDateTimeUTC()`

Parses a string and returns a JavaScript Date.

Signature

```
parseDateTime(string dateTimeString, string parseFormat, string | boolean locale,
boolean strictParsing)
```

Parameters

`dateTimeString`
Type: `string`

A datetime string.

`parseFormat`
Type: `string`

An optional Java format string used to parse the datetime. The default is from the `$Locale` global value provider.

`locale`

Type: `string | boolean`

This parameter is deprecated.

`strictParsing`

Type: `string`

Set this optional parameter to `true` to turn off forgiving parsing and use strict validation.

Returns

Type: `Date`

Returns a JavaScript `Date` object, or `null` if `dateTimeString` is invalid.

Sample Code

```
var date = "2015-10-9";
// Returns "Thu Oct 08 2015 17:00:00 GMT-0700 (PDT)"
$A.localizationService.parseDateTimeUTC(date);
```

`startOf()`

Returns a date that is the start of a unit of time for the given date.

Signature

`startOf(string | number | Date date, string unit)`

Parameters

`date`

Type: `string | number | Date`

A datetime string in ISO8601 format, or a timestamp in milliseconds, or a `Date` object.

`unit`

Type: `string`

A datetime unit. Options are 'year', 'month', 'week', 'day', 'hour', 'minute', or 'second'.

Returns

Type: `Date`

A JavaScript `Date` object. If a unit is not provided, returns a parsed date.

Sample Code

```
var date = "2015-10-9";
// Returns "Thu Oct 01 2015 00:00:00 GMT-0700 (PDT)"
$A.localizationService.startOf(date, 'month');
```

`toISOString()`

Deprecated. Use `Date.toISOString()` instead.

Signature

`toISOString(Date | T date)`

Parameters

`date`
Type: `Date | T`

A `Date` object.

`unit`
Type: `string`

A datetime unit. Options are 'year', 'month', 'week', 'day', 'hour', 'minute', or 'second'.

Returns

Type: `Date`
An ISO8601 string.

`translateFromLocalizedDigits()`

Translate the localized digit string to a string with Arabic digits, if there is any.

Signature

`translateFromLocalizedDigits(string input)`

Parameters

`input`
Type: `string`

A string with localized digits.

Returns

Type: `string`
A string with Arabic digits.

`translateFromOtherCalendar()`

Translates the input date from another calendar system (for example, the Buddhist calendar) to the Gregorian calendar based on the locale.

Signature

`translateFromOtherCalendar(Date date)`

Parameters

`date`
Type: `Date`

A `Date` object.

Returns

Type: `Date`
Returns a translated `Date` object.

`translateToLocalizedDigits()`

Translate the input string to a string with localized digits, if there is any.

Signature

`translateToLocalizedDigits(string input)`

Parameters

`input`
Type: `string`

A string with Arabic digits.

Returns

Type: `string`
A string with localized digits.

`translateToOtherCalendar()`

Translates the input date to a date in another calendar system (for example, the Buddhist calendar) based on the locale.

Signature

`translateToOtherCalendar(Date date)`

Parameters

`date`
Type: `Date`

A `Date` object.

Returns

Type: `Date`

Returns a translated `Date` object.

Component

`Component` contains methods to work with components.

Methods

IN THIS SECTION:

addEventHandler()

Dynamically adds an event handler for a component or application event.

addHandler()

Deprecated. Use `addEventHandler()` instead.

addValueHandler()

Adds handlers to values owned by the component.

addValueProvider()

Adds custom value providers to a component.

autoDestroy()

Sets a flag to tell the rendering service whether or not to destroy this component when it is removed from its rendering facet.

clearReference()

Clears a live reference for the value passed in using property syntax. For example, if you use `aura:set` to set a value and later want to reset the value using `component.set()`, clear the reference before resetting the value.

destroy()

Destroys the component and cleans up memory. After a component that is declared in markup is no longer in use, the framework automatically destroys it and frees up its memory. If you create a component dynamically in JavaScript and that component isn't added to a facet (`v.body` or another attribute of type `Aura.Component[]`), you have to destroy it manually using `destroy()` to avoid memory leaks.

find()

Locates a component using its local ID (`aura:id`).

get()

Returns the value referenced using property syntax. For example, `cmp.get("v.attr")` returns the value of the `attr` attribute.

getConcreteComponent()

Gets the concrete implementation of a component. If the component is concrete, the method returns the component itself. For example, call this method to get the concrete component of a super component.

getElement()

If the component renders only a single element, return it. Otherwise, use `getElements()`.

getElements()

Returns a map of the elements rendered by the component.

getEvent()
Returns a new event instance of the named component event.

getGlobalId()
Gets the global ID, which is the generated globally unique id of the component. It can be used to locate the instance later, but will change across page loads.

getLocalId()
Gets the ID set using the `aura:id` attribute. Pass the local ID into `find()` on the parent component to locate this child component.

getName()
Returns the component's code-compatible camel case name, such as `'lightningButton'`.

getReference()
Returns a live reference to the value indicated using property syntax. This method is useful when you dynamically create a component.

getSuper()
Returns the super component.

getType()
Returns the component's canonical type; for example, `'lightning:button'`.

getVersion()
Returns the component's version number.

isConcrete()
Returns `true` if the component is concrete, or `false` otherwise. A concrete component is a sub-component in an inheritance chain.

isInstanceOf()
Checks whether a component is an instance of the given component or interface name.

isValid()
Returns `true` if the component has not been destroyed.

removeEventHandler()
Dynamically removes a component event handler for the specified event.

set()
Sets the value referenced using property syntax.

addEventHandler()

Dynamically adds an event handler for a component or application event.

Signature

```
addEventHandler(String event, function handler, String phase, Boolean includeFacets)
```

Parameters

event
Type: `String`

The name of the event to handle. For a component event, set this argument to match the name attribute of the `aura:registerEvent` tag. For an application event, set this argument to match the event descriptor, `namespace:eventName`.

`handler`
Type: `function`

The handler for the event. There are two format options for this argument.

- To use a controller action, use the format: `cmp.getReference("c.actionName")`.
- To use an anonymous function, use the format: `function(auraEvent) { // handling logic here }`

`phase`
Type: `String`

Optional. The event bubbling phase for which to add the handler. The default value is `"bubble"`.

`includeFacets`
Type: `Boolean`

If `true`, attempts to catch events generated by components transcluded by facets; for example `v.body`.

Sample Code

```
// For component event, first param matches name attribute in <aura:registerEvent> tag
cmp.addEventHandler("compEvent", cmp.getReference("c.handleEvent"));

// For application event, first param is event descriptor, "c:appEvent"
cmp.addEventHandler("c:appEvent", cmp.getReference("c.handleAppEvent"));

// Anonymous function handler for component event
cmp.addEventHandler("compEvent", function(auraEvent) {
    // add handler logic here
    console.log("Handled the component event in anonymous function");
});
```

SEE ALSO:

Dynamically Adding Event Handlers To a Component

removeEventHandler()

addHandler()

Deprecated. Use `addEventHandler()` instead.

Signature

`addHandler(String eventName, Object valueProvider, Object actionExpression, Boolean insert, String phase, Boolean includeFacets)`

Parameters

`eventName`
Type: `String`

The name of the event to handle. For a component event, set this argument to match the name attribute of the `aura:registerEvent` tag. For an application event, set this argument to match the event descriptor, `namespace:eventName`.

`valueProvider`
Type: `Object`

The value provider to use for resolving the `actionExpression`.

`actionExpression`
Type: `Object`

The expression to use for resolving the handler action against the given `valueProvider`.

`insert`
Type: `Boolean`

If `true`, put the handler at the beginning instead of the end of the handler array.

`phase`
Type: `String`

Optional. The event bubbling phase for which to add the handler. The default value is `"bubble"`.

`includeFacets`
Type: `Boolean`

If `true`, attempts to catch events generated by components transcluded by facets; for example `v.body`.

SEE ALSO:
addEventHandler()

`addValueHandler()`

Adds handlers to values owned by the component.

Signature

`addValueHandler(Object config)`

Parameters

`config`
Type: `Object`

The value event, such as `"change"`, and the action, such as `"c.myAction"`.

`addValueProvider()`

Adds custom value providers to a component.

Signature

`addValueProvider(String key, Object valueProvider)`

Parameters

`key`

Type: `String`

Key to identify the value provider. Used in expressions in markup.

`valueProvider`

Type: `Object`

The object to request data from. Must implement `get(expression)`, can implement `set(key,value)`.

SEE ALSO:

Value Providers

`autoDestroy()`

Sets a flag to tell the rendering service whether or not to destroy this component when it is removed from its rendering facet.

Signature

`autoDestroy(Boolean destroy)`

Parameters

`destroy`

Type: `Boolean`

Default is `true`, which marks the component to be destroyed when it's orphaned. Set to `false` to keep a reference to a component after it has been unrendered or removed from a parent facet. We don't recommend setting the value to `false`. If you do, be careful to avoid memory leaks.

`clearReference()`

Clears a live reference for the value passed in using property syntax. For example, if you use `aura:set` to set a value and later want to reset the value using `component.set()`, clear the reference before resetting the value.

Signature

`clearReference(String key)`

Parameters

`key`

Type: `String`

The data key for which to clear the reference. For example, `"v.attributeName"`.

`destroy()`

Destroys the component and cleans up memory. After a component that is declared in markup is no longer in use, the framework automatically destroys it and frees up its memory. If you create a component dynamically in JavaScript and that component isn't added to a facet (`v.body` or another attribute of type `Aura.Component[]`), you have to destroy it manually using `destroy()` to avoid memory leaks.

Signature

`destroy()`

`find()`

Locates a component using its local ID (`aura:id`).

Returns different types depending on the result.

1. If the local ID is unique, returns the component.
2. If there are multiple components with the same local ID, returns an array of the components.
3. If there is no matching local ID, returns `undefined`.

Signature

`find(String | Object name)`

Parameters

`name`

Type: `String | Object`

If name is an object, return instances of it. Otherwise, finds a component using its `aura:id`.

SEE ALSO:

Finding Components by ID

`get()`

Returns the value referenced using property syntax. For example, `cmp.get("v.attr")` returns the value of the `attr` attribute.

Signature

`get(String key)`

Parameters

`key`

Type: `String`

The data key to look up on the component.

`getConcreteComponent()`

Gets the concrete implementation of a component. If the component is concrete, the method returns the component itself. For example, call this method to get the concrete component of a super component.

Signature

`getConcreteComponent()`

SEE ALSO:

Favor Composition Over Inheritance

`getElement()`

If the component renders only a single element, return it. Otherwise, use `getElements()`.

Signature

`getElement()`

`getElements()`

Returns a map of the elements rendered by the component.

Signature

`getElements()`

`getEvent()`

Returns a new event instance of the named component event.

Signature

`getEvent(String name)`

Parameters

`name`

Type: `String`

The name of the event.

Sample Code

```
// evtName matches the name attribute in aura:registerEvent
cmp.getEvent("evtName");
```

`getGlobalId()`

Gets the global ID, which is the generated globally unique id of the component. It can be used to locate the instance later, but will change across page loads.

Signature

`getGlobalId()`

SEE ALSO:

Component IDs

`getLocalId()`

Gets the ID set using the `aura:id` attribute. Pass the local ID into `find()` on the parent component to locate this child component.

Signature

`getLocalId()`

SEE ALSO:

find()

`getName()`

Returns the component's code-compatible camel case name, such as `'lightningButton'`.

Signature

`getName()`

Returns

Type: `String`

The component name.

`getReference()`

Returns a live reference to the value indicated using property syntax. This method is useful when you dynamically create a component.

Signature

`getReference(String key)`

Parameters

`key`

Type: `String`

The data key for which to return a reference.

Returns

Type: `PropertyReferenceValue`
A property reference value.

`getSuper()`

Returns the super component.

Signature

`getSuper()`

Returns

Type: `Component`
The super component.

`getType()`

Returns the component's canonical type; for example, `'lightning:button'`.

Signature

`getType()`

Returns

Type: `String`
The component's type.

`getVersion()`

Returns the component's version number.

Signature

`getVersion()`

Returns

Type: `String`
The component name.

`isConcrete()`

Returns `true` if the component is concrete, or `false` otherwise. A concrete component is a sub-component in an inheritance chain.

Signature

`isConcrete()`

Returns

Type: `Boolean`

Returns `true` if the component is concrete, or `false` otherwise.

SEE ALSO:

getConcreteComponent()

Favor Composition Over Inheritance

`isInstanceOf()`

Checks whether a component is an instance of the given component or interface name.

Signature

`isInstanceOf(String name)`

Parameters

`name`

Type: `String`

The name of the component or interface, with a format of `namespace:componentName`.

Returns

Type: `Boolean`

Returns `true` if the component is an instance, or `false` otherwise.

`isValid()`

Returns `true` if the component has not been destroyed.

Signature

`isValid()`

Returns

Type: `Boolean`

Returns `true` if the component has not been destroyed, or `false` otherwise.

`removeEventHandler()`

Dynamically removes a component event handler for the specified event.

Signature

`removeEventHandler(String event, function handler, String phase)`

Parameters

`event`
: Type: `String`

  The name of the event to remove; for example, `'c:myEvent'`.

`handler`
: Type: `function`

  A reference to the function or action to remove; for example., `'cmp.getReference("c.handleMyEvent");'`.

`phase`
: Type: `String`

  Optional. The event bubbling phase for which to remove the handler. The default value is `"default"`.

SEE ALSO:

addEventHandler()

`set()`

Sets the value referenced using property syntax.

Signature

`set(String key, Object value)`

Parameters

`key`
: Type: `String`

  The data key to set on the component; for example, `cmp.set("v.key","value")`.

`value`
: Type: `Object`

  The value to set.

SEE ALSO:

get()

Event

`Event` contains methods to work with events. Use an event to communicate between components.

Methods

IN THIS SECTION:

fire()
Fires an event.

getEventType()
Returns the type of the event. Possible values are `'COMPONENT'` or `'APPLICATION'`.

getName()
Returns an event's name.

getParam()
Returns the value of an event's parameter.

getParams()
Returns the value of all an event's parameters.

getPhase()
Returns the current phase of an event. Returns `undefined` if the event hasn't been fired yet. Possible return values for application and component events are `"capture"`, `"bubble"`, and `"default"` once fired. A value event returns `"default"` once it's fired.

getSource()
Returns the source component that fired an event.

getSourceEvent()
Returns the source event that fired this event, if it was fired by an event binding, such as `{!e.myEvent}`.

getType()
Returns the type of the event's definition, such as `'c:myEvent'`.

pause()
Pauses an event. Event handlers aren't processed until `Event.resume()` is called. The handling process pauses in the current position of the event handler processing sequence. If the event is already paused, this method does nothing. This method throws an error if it's called in the `"default"` phase.

preventDefault()
Prevents the default phase execution for this event. This method throws an error if it's called in the `"default"` phase.

resume()
Resumes event handling for this event from the same position in the event handler processing sequence from which it was previously paused. If the event isn't paused, this method does nothing. This method throws an error if it's called in the `"default"` phase. Any remaining event handlers might execute in the current call stack or might be deferred and executed in a new call stack. Therefore, the exact timing behavior is not predictable.

setParam()
Sets a parameter for an event. This method doesn't modify an event that has already been fired.

setParams()
Sets parameters for an event. This method doesn't modify an event that has already been fired.

stopPropagation()
Sets whether the event can bubble or not. This method throws an error if called in the `"default"` phase.

`fire()`

Fires an event.

Signature

`fire(Object params)`

Parameters

`params`
: Type: `Object`

  An optional set of parameters for the event. Any previous parameters of the same name are overwritten.

`getEventType()`

Returns the type of the event. Possible values are `'COMPONENT'` or `'APPLICATION'`.

Signature

`getEventType()`

Returns

Type: `String`
: The event type.

`getName()`

Returns an event's name.

Signature

`getName()`

Returns

Type: `String`
: The event name.

`getParam()`

Returns the value of an event's parameter.

Signature

`getParam(String name)`

Parameters

`name`
: Type: `String`

The parameter name. For example, `event.getParam("button")` returns the value of the pressed mouse button (0, 1, or 2).

Returns

Type: `Object`
: The parameter value.

`getParams()`

Returns the value of all an event's parameters.

Signature

`getParams()`

Returns

Type: `Object`
: The collection of parameters.

`getPhase()`

Returns the current phase of an event. Returns `undefined` if the event hasn't been fired yet. Possible return values for application and component events are `"capture"`, `"bubble"`, and `"default"` once fired. A value event returns `"default"` once it's fired.

Signature

`getPhase()`

Returns

Type: `String`
: The current phase of the event.

`getSource()`

Returns the source component that fired an event.

Signature

`getSource()`

Returns

Type: `Object`

The source component that fired the event.

`getSourceEvent()`

Returns the source event that fired this event, if it was fired by an event binding, such as `{!e.myEvent}`.

Signature

```
getSourceEvent()
```

Returns

Type: `Object`

The source event that fired the event.

`getType()`

Returns the type of the event's definition, such as `'c:myEvent'`.

Signature

```
getType()
```

Returns

Type: `String`

The event definition type.

`pause()`

Pauses an event. Event handlers aren't processed until `Event.resume()` is called. The handling process pauses in the current position of the event handler processing sequence. If the event is already paused, this method does nothing. This method throws an error if it's called in the `"default"` phase.

Signature

```
pause()
```

`preventDefault()`

Prevents the default phase execution for this event. This method throws an error if it's called in the `"default"` phase.

Signature

```
preventDefault()
```

`resume()`

Resumes event handling for this event from the same position in the event handler processing sequence from which it was previously paused. If the event isn't paused, this method does nothing. This method throws an error if it's called in the `"default"` phase. Any remaining event handlers might execute in the current call stack or might be deferred and executed in a new call stack. Therefore, the exact timing behavior is not predictable.

Signature

`resume()`

`setParam()`

Sets a parameter for an event. This method doesn't modify an event that has already been fired.

Signature

`setParam(String key, Object value)`

Parameters

`key`
Type: `String`

The name of the parameter.

`value`
Type: `Object`

The value of the parameter.

`setParams()`

Sets parameters for an event. This method doesn't modify an event that has already been fired.

Signature

`setParams(Object config)`

Parameters

`config`
Type: `Object`

The event's parameter.

`stopPropagation()`

Sets whether the event can bubble or not. This method throws an error if called in the `"default"` phase.

Signature

`stopPropagation()`

Util

`Util` contains utility methods.

Methods

IN THIS SECTION:

`addClass()`

Adds a CSS class to a component.

Signature

`addClass(Object element, String newClass)`

Parameters

element
: Type: `Object`

  The component to apply the class on.

newClass
: Type: `String`

  The CSS class to be applied.

Sample Code

```
// find a component with aura:id="myCmp" in markup
var myCmp = component.find("myCmp");
$A.util.addClass(myCmp, "myClass");
```

`getBooleanValue()`

Coerces truthy and falsy values into a native boolean.

Signature

`getBooleanValue(Object val)`

Parameters

val
: Type: `Object`

  The object to check.

Returns

Type: `String`
: Returns `true` if the object is truthy, or `false` otherwise.

`hasClass()`

Checks whether the component has the specified CSS class.

Signature

`hasClass(Object element, String className)`

Parameters

element
: Type: `Object`

  The component to check.

className
  Type: `String`

  The CSS class name to check for.

Returns

Type: `Boolean`
  Returns `true` if the specified class is found for the component, or `false` otherwise.

Sample Code

```
// find a component with aura:id="myCmp" in markup
var myCmp = component.find("myCmp");
$A.util.hasClass(myCmp, "myClass");
```

`isArray()`

Checks whether the specified object is an array.

Signature

`isArray(Object obj)`

Parameters

obj
  Type: `Object`

  The object to check.

Returns

Type: `Boolean`
  Returns `true` if the object is an array, or `false` otherwise.

`isEmpty()`

Checks if the object is empty. An empty object's value is `undefined`, `null`, an empty array, or an empty string. An object with no native properties is not considered empty.

Signature

`isEmpty(Object obj)`

Parameters

obj
  Type: `Object`

  The object to check.

Returns

Type: `Boolean`

Returns `true` if the object is empty, or `false` otherwise.

`isObject()`

Checks whether the specified object is a valid object. A valid object is not a DOM element, is not a native browser class (`XMLHttpRequest`) is not falsey, and is not an array, error, function string or a number.

Signature

```
isObject(Object obj)
```

Parameters

`obj`

Type: `Object`

The object to check.

Returns

Type: `Boolean`

Returns `true` if the object is a valid object, or `false` otherwise.

`isUndefined()`

Checks if the object is `undefined`.

Signature

```
isUndefined(Object obj)
```

Parameters

`obj`

Type: `Object`

The object to check.

Returns

Type: `Boolean`

Returns `true` if the object is undefined, or `false` otherwise.

`isUndefinedOrNull()`

Checks if the object is `undefined` or `null`.

Signature

`isUndefinedOrNull(Object obj)`

Parameters

`obj`
: Type: `Object`

  The object to check.

Returns

Type: `Boolean`
: Returns `true` if the object is `undefined` or `null`, or `false` otherwise.

`removeClass()`

Removes a CSS class from a component.

Signature

`removeClass(Object element, String newClass)`

Parameters

`element`
: Type: `Object`

  The component to remove the class from.

`newClass`
: Type: `String`

  The CSS class to be removed.

Sample Code

```
//find a component with aura:id="myCmp" in markup
var myCmp = component.find("myCmp");
$A.util.removeClass(myCmp, "myClass");
```

`toggleClass()`

Toggles (adds or removes) a CSS class from a component.

Signature

`toggleClass(Object element, String className)`

Parameters

`element`

Type: `Object`

The component to add or remove the class from.

`className`

Type: `String`

The CSS class to be added or removed.

Sample Code

```
// find a component with aura:id="toggleMe" in markup
var toggleText = component.find("toggleMe");
$A.util.toggleClass(toggleText, "toggle");
```

INDEX