Visualforce Performance: Best Practices

Salesforce, Winter '21

@salesforcedocs

Last updated: August 20, 2020

CONTENTS

VISUALFORCE PERFORMANCE: BEST PRACTICES

Preface

This document contains best practices for optimizing the performance of Visualforce pages. Its contents are a subset of the best practices in the *Visualforce Developer's Guide*.

Audience

This document is for developers and architects implementing Visualforce pages.

Assumptions

This document assumes a basic understanding of Visualforce and related Lightning Platform technologies.

Methods of Locating and Testing for Performance Issues

Before following the best practices outlined in this paper:

- Determine the cause(s) of the performance issues.
- Ensure that your pages follow general best practices for Web design, if Visualforce isn't causing the performance issues.
- Test the scope and impact of the performance issues.

Investigating Visualforce Performance Issues

Use the Lightning Platform Developer Console to investigate the performance of your Visualforce markup and other Lightning Platform features on the page.

The Developer Console has a debug log that details the performance of requests as the server processes them. The details show every execution step for methods, queries, workflows, callouts, DML, validations, triggers, and pages by type and time, helping you:

- See what's consuming system resources
- Identify issues in your code

See Working with the Developer Console.

Verifying General Web Design Best Practices

If Visualforce isn't causing your performance issues, verify that your pages follow general best practices for Web design, such as:

- JavaScript and CSS minification
- Image optimization for the Web
- Avoidance of iframes, whenever possible

In addition, use HTML, CSS, and JavaScript profiling and debugging tools to provide insight into network latency and load times, the efficiency of JavaScript code, and more.

Google Chrome™ Developer Tools

Use the Chrome Developer Tools to see what's happening behind the scenes of the Chrome™ browser and the application you're running within it.

Firebug

Use Firebug, an add-on for Mozilla® Firefox®, to work with and monitor the following—all while keeping your current, live Web page open.

- CSS
- HTML
- JavaScript

YSlow

Use YSlow, an add-on from Yahoo!®, to receive automated suggestions for improving Web pages.

Note: You must already have Firebug installed to use YSlow.

WebPagetest

Use WebPagetest to gauge how a Web page performs according to the following criteria.

- Test location
- Browser
- Connection speed

Testing Visualforce Pages

If your Visualforce page has performance issues, test it on multiple machines and with multiple browsers to ensure that the problem isn't confined to a single user's computer. In addition, check the load time of other Salesforce pages and other websites. If Salesforce pages load slowly, check the status of the Salesforce servers at http://trust.salesforce.com/trust/status/. If all Web pages load slowly, check your network configuration.

To test for and help prevent performance regressions:

- Use tools like HP LoadRunner and Selenium to automate the testing of tedious or complex flows that might produce inconsistent results. Automated tests can click links, enter and retrieve data, and record execution times. They might uncover bottlenecks and defects missed by manual testing.
- Test in as many browsers and with as many versions as possible.
- Test with large data volumes, even if your pages avoid unbounded data, implement pagination, and display only relevant data. These tests might reveal scenarios with data skews where certain users have access to too many records.
- Test on actual mobile devices to uncover performance issues that wouldn't be apparent on developer machines. Mobile clients have different performance profiles because of slower processors, limited memory, and slower network connections.

Tip: Use tools like WebPagetest for initial mobile browser testing and use actual devices for in-depth testing.

Best Practices for Optimizing Visualforce Performance

Consider following these best practices to improve the performance of your Visualforce pages.

- Design your Visualforce pages according to some general guidelines.
- Use standard objects and declarative features.
- Limit the amount of data that your Visualforce pages display.

- Delay expensive calculations or data loading.
- Offload processing to asynchronous tasks.
- Cache global data in custom settings.
- Write efficient:
  - Apex and SOQL
  - Getter methods
- Optimize:
  - View state
  - Component hierarchies
  - Polling
  - HTML
  - CSS
  - JavaScript
  - Image usage
  - Pages for Internet Explorer

Following General Design Guidelines for Application Performance

When designing Visualforce pages, follow these general guidelines to avoid performance impacts.

- Design pages around specific tasks, with a sensible workflow and navigation between tasks.
- Don't overload pages with functionality and data. Visualforce pages with unbounded data or a large number of components, rows, and fields have poor usability and performance, and they risk hitting governor limits for view state, heap size, record limits, and total page size.
- Push back on requests to include nonessential functionality.
- Build prototypes to validate concerns.

Using Standard Functionality vs. Creating Custom Code

The programmatic features of the Lightning Platform platform make it easy to customize functionality, but always use standard objects and declarative features whenever possible. Standard objects and declarative features—such as approval processes, flows, and workflow ruless—are highly optimized already and don't count against most governor limits. They typically simplify data models and reduce the number of Visualforce pages necessary for business processes.

Controlling Data Size

Visualforce pages can't exceed the 15 MB standard response limit, but even smaller page sizes affect load time.

To minimize load times, use the following techniques to further limit the amount of data each page displays.

Filters

Use filters to limit the data that SOQL calls in and Apex controllers return.

For example, use `AND` statements in your `WHERE` clause or remove null results.

Keywords

When creating Apex controllers, use the `with sharing` keyword to retrieve only the records the user can access.

`StandardSetController` Built-In Pagination

Use the built-in pagination functionality in list controllers to prevent list views from displaying unbounded data. Unbounded data might cause longer load times, hit governor limits, and become unusable as the data set grows. By default, a list controller returns 20 records on the page, but developers often configure list views to display up to 100 records at a time.

Tip: To control the number of records each page displays, use a controller extension to set the `pageSize`.

`SOQL OFFSET` Pagination

Use the `SOQL OFFSET` clause to write logic that paginates to a specific subset of results within SOQL.

In addition, avoid using data grids, which display many records with editable fields. Data grids frequently expand to thousands of input components on a page and exceed the maximum view state size, resulting in a Visualforce component tree that's slow to process.

If your Visualforce page needs a data grid:

- Use pagination and filters.
- Where possible, make data read-only to reduce the view state size.
- Only display essential data for a given record, and provide a link for an Ajax-based details box or separate page to view the rest.

Lazy Loading

To reduce or delay expensive calculations or data loading, use *lazy loading*, a technique in which a page loads its essential features first and delays the rest—either briefly or until the user performs an action that requires the information. This technique gives users faster access to essential features, improving the apparent responsiveness of a large page, even though the entire page takes the same total time to load.

To lazy load parts of a Visualforce page:

- Use the `rerender` attribute on Visualforce components to update the component without updating the entire page.
- Use JavaScript Remoting to call functions in your controller through JavaScript, and to retrieve ancillary or static data.
- Create a custom component to show and hide data according to user actions.

When lazy loading pages, consider the number of users and amount of data you expect to use the page, and watch out for limits like the concurrent API call limit. For example, if a navigation tree only loads elements as needed, the number of queries might end up out of proportion to the data.

Offloading Processing to Asynchronous Tasks

Offload expensive processing using asynchronous tasks when that processing is secondary to the purpose of the pages. For example, instead of having a user click a button and wait for a long-running task to complete before seeing a confirmation message, consider queuing the long-running task for asynchronous processing and returning control to the user immediately. Configure the page to notify the user via email or some other means when the task completes.

Caching Global Data in Custom Settings

Visualforce pages sometimes use calculation results globally; that is, the pages use the same data across users and requests. Improve the performance of such pages by caching calculation results in a custom setting and refreshing the results periodically instead of upon every request. Custom settings are part of an application's cache and do not require a database query for retrieval. Balance this approach against the time it takes to update custom cached data.

Writing Efficient Apex and SOQL

The Apex and SOQL that a Visualforce page leverages impacts the overall performance of the page.

When writing Apex or SOQL for use with a Visualforce page:

- Perform calculations in SOQL instead of in Apex, whenever possible.
- Never perform DML inside a loop.
- Filter in SOQL first, then in Apex, and finally in Visualforce.

  For example, don't do the following.

```
<apex:pageBlockTable value="{!positions}" var="position">
    <apex:column value="{!position.name}"
         rendered="{!IF(position.Status__c == 'Open - Approved', true, false)}"/>
    <apex:column value="{!position.Location__c}"
         rendered="{!IF(position.Status__c == 'Open - Approved', true, false)}"/>
    <apex:column value="{!position.Job_Description__c}"
         rendered="{!IF(position.Status__c == 'Open - Approved', true, false)}"/>
</apex:pageBlockTable>
```

See the Apex Developer Guide and Best Practices for Deploying Large Data Volumes.

Writing Efficient Getter Methods

Visualforce requests evaluate expressions, action attributes, and other method calls, sometimes calling the getter methods in a class multiple times, especially in postbacks (form submissions). Cache the value of a property calculation so that additional calls to access the property can use the cached value, and configure the getters in your Apex classes to only query for data if the object is null.

Optimizing the View State

To maintain state in a Visualforce page, the Lightning Platform platform includes the state of components, field values, and controller state in a hidden form element. This encrypted string is the *view state* and has a limit of 170KB. Large view states require longer processing times for each request, including serializing and deserializing, and encryption and decryption. Reducing your view state size causes your pages to load quicker and stall less often.

Use the **View State** tab in the development mode footer to monitor view state performance, and take the following actions.

- Use the `transient` keyword in your Apex controllers for variables that aren't essential for maintaining state and aren't necessary during page refreshes.
- If you notice that a large percentage of your view state comes from objects used in controllers or controller extensions, consider refining your SOQL calls to return only data that's relevant to the Visualforce page.
- If your view state is affected by a large component tree, try reducing the number of components your page depends on.

To reduce view state:

- Use filters and pagination to reduce data requiring state.
- Declare instance variables with a transient keyword when the variable is only useful for the current request. The view state includes all non-transient members in the controller and extension, as well as objects reachable from these non-transient members. Decide if some data can be read-only and use the `<apex:outputText>` component instead of `<apex:inputField>`.
- Set the `Development Mode` and `Show View State in Development Mode` permissions set to see the **View State** tab in the development mode footer. The tab displays the distribution of view state. Make sure you know the view state size of each page, and test with large data volumes to determine if issues might occur after deployment.

- Use JavaScript remoting. Unlike the `<apex:actionFunction>` component, JavaScript Remoting does not require a form component. This technique doesn't reduce the overall view state of a page, but your page generally performs better without the need to transmit, serialize, and deserialize the view state. The tradeoff is the loss of the re-render attribute and the additional JavaScript code to handle callbacks.

See https://developer.salesforce.com/page/An_Introduction_to_Visualforce_View_State.

Optimizing Component Hierarchies

Flat component structures process faster than deep, hierarchical component structures, so limit the nesting of custom components to logically organize functionality, and leverage custom components only when that logic is intended for reuse or inclusion in another package. Vast hierarchies increase server-side management and processing time because Visualforce maintains context throughout the entire request, and traversing component hierarchies consumes time and resources. Deeply nested components incur the highest processing costs. Vast hierarchies also put pages at risk of hitting hit heap size governor limits.

Optimizing Polling

The `<apex:actionPoller>` component is a timer that makes Ajax requests. Pages that use the `<apex:actionPoller>` component make continuous requests on the server. These concurrent requests might be long-running transactions that block other pending transactions. If a user leaves the page open for long periods of time or opens multiple windows on the same page—for example, to get details for multiple accounts—performance decreases.

To reduce the load on the server and avoid hitting governor limits, increase the value of the `interval` attribute on the `<apex:actionPoller>` component to increase the interval at which it calls Apex from your Visualforce page.

For example, adjust the `interval` attribute to 15 seconds instead of 5. In addition, move any non-essential logic to an asynchronous code block using Ajax., and test the page with simultaneous tabs or windows open to check if there are multiple polls coming from the same computer.

The `<apex:actionPoller>` component is appropriate on pages that don't require expensive processing, but for pages where the calculations require more server time, consider using the `<apex:actionFunction>` component with JavaScript Remoting instead. This alternative requires more code but offers greater flexibility and efficiency.

See *Two Visualforce Pages: ActionFunction and JavaScript Remoting*.

Optimizing HTML

Optimize your Visualforce page HTML for efficient processing, both on the server side where Visualforce validates it for correctness, and on the client side where it makes performance more responsive in the user's browser.

- Review the HTML that Visualforce components generate. Visualforce pages require valid HTML and might correct invalid HTML during compilation, causing your HTML to render in ways you don't intend. For example, if you have a `<head>` or `<body>` tag inside of your `<apex:page>` tag, the Visualforce page removes it at runtime.
- Review Ajax code. During an Ajax request, the server validates and corrects inbound HTML to ensure that the response properly fits back into the DOM. This process is quick if your Visualforce page contains valid markup and if corrections are unnecessary.
- Reduce HTML bloat. Although the browser caches the HTML and compiled Visualforce tags, retrieving them from the cache impacts performance. Unnecessary HTML also increases the size of the component tree and the processing time for Ajax requests.

Optimizing CSS

While CSS styles can improve the appearance of Visualforce pages, they might also add significant weight.

Follow these tips to optimize your Visualforce page CSS for efficient delivery to the client, improve caching, and accelerate page display in the browser.

- Consider externalizing stylesheets, which involves taking styles out of the page itself and putting them in separate CSS files. This practice increases the number of initial HTTP requests but reduces the size of individual pages. After the browser caches the stylesheets, the overall request size decreases.
- Combine all CSS files into a single file to reduce the number of HTTP requests.
- Remove comments and whitespace (spaces, newlines, and tabs), and compress the resulting file for faster downloads.
- Use static resources to serve CSS files, as well as images, JavaScript, and other non-changing files. Stylesheets and other assets served this way benefit from the caching and content distribution network (CDN) built into Salesforce.
- For pages that don't use Salesforce CSS files, set the `<apex:page>` tag's `showHeaders` and `standardStylesheets` attributes to `false`. This practice excludes the standard Salesforce CSS files from the generated page header.

Tip: For Javascript and CSS, it might be burdensome during development to make a change, process and package it, and deploy. Consider automating this process through a script.

Optimizing JavaScript

Optimize your JavaScript to ensure efficient delivery to the client, improve caching, and accelerate page display in the browser.

- Consider externalizing JavaScript files. This process increases the number of initial HTTP requests, but it also reduces the size of individual pages and takes advantage of browser caching.
- Build custom versions of JavaScript libraries with only the functions you need. Many open-source JavaScript libraries, such as jQuery, provide this option, which significantly reduces the file size.
- Combine all JavaScript files into a single file to reduce HTTP requests, and remove duplicate functions as they might result in more than one HTTP request and waste JavaScript execution.
- Remove comments and whitespace (spaces, newlines, and tabs), and compress the resulting file for faster downloads.
- Put scripts at the bottom of the page. By loading scripts just before the closing `</body>` tag, the page can download other components first and render the page progressively.

  Note: Only move JavaScript to the bottom of the page if you're certain it doesn't have any adverse effects on your page. For example, do not move JavaScript code snippets requiring `document.write` or event handlers from the `<head>` element.

- Consider including JavaScript files using a standard HTML `<script>` tag right before your closing `</apex:page>` tag instead of using `<apex:includeScript>`.

  The `<apex:includeScript>` tag places JavaScript right before the closing `</head>` element, causing the browser to attempt to load the JavaScript before rendering any other content on the page.

- Use static resources to serve JavaScript files, as well as images, CSS, and other non-changing files. JavaScript and other assets served this way benefit from the caching and content distribution network (CDN) built into Salesforce.

Tip: For Javascript and CSS, it might be burdensome during development to make a change, process and package it, and deploy. Consider automating this process through a script.

Optimizing Image Usage

Images are frequently the largest components of a Web page and thus significantly affect performance.

Follow these image tips to minimize the performance impact of images.

- Use fewer images and smaller background textures, and use CSS instead of images whenever possible.

- Use *CSS sprites* instead of individual images. CSS sprites let you combine a collection of similarly sized graphics, such as buttons and icons, into a single file, and then use the CSS `background-image` and `background-position` properties to display portions of the combined image. This technique reduces the number of images used—and thus the number of HTTP requests—and the combined sprite file of images is usually very easily cached.
- Use static resources to serve images, as well as CSS, JavaScript, and other non-changing files. Images and other assets served this way benefit from the caching and content distribution network (CDN) built into Salesforce.
- Compress images. Graphics tools often use default settings that favor visual fidelity over compression and add metadata when saving images. Image compression tools can compress image files 10-30% further without reducing visual quality.

  Tip: Consider adding a script that compresses image assets to your development workflow.

Optimizing Pages for Internet Explorer

If your organization uses Microsoft® Internet Explorer®, optimize your Visualforce pages as follows.

- Avoid the `AlphaImageLoader` filter. It blocks rendering and freezes the browser while the image downloads. It also increases memory consumption and applies per element, not per image, so the problem multiplies.
- To include additional stylesheets, use `<link>` instead of `@import`. Using `@import` changes the download order and hinders progressive rendering.
- Avoid CSS expressions. While they allow you to set CSS properties dynamically, Internet Explorer evaluates them more often than expected.

Visualforce Performance Optimization Case Study

To understand how these best practices work together in actual situations, consider a case in which a customer had a Visualforce page with a data grid to collect sales forecasts. The data model for the forecast contained a multilevel object hierarchy. The page also contained calculations to display pivoted data. For an average user, the grid contained roughly 1,500 cells, causing the page to load slowly, and hit heap and view state limits.

If you are in a similar situation and want to optimize page performance:

- Make the page task focused, using it only for input. Using the page for both input and some aggregate reporting adds unnecessary complexity.
- Create a custom object to hold aggregate data for reporting. Removing the formulas needed to display aggregated information reduces the heap size.
- Relax some of the design requirements, most notably pagination.
- Avoid displaying every account on a single page. This practice improves both page load speeds and the view state size.
- Make the data grid cells read-only. Have users click on cells to edit them, and use Ajax to save their edits. These practices have a big impact on view state.

Other Resources and Tools

For more information on Web page performance optimization, see:

- Steve Souders' High Performance Web Sites blog
- Exceptional Performance team - Yahoo! Developer Network
- *Two Visualforce Pages: ActionFunction and JavaScript Remoting*

- *An Introduction to Visualforce View State*