

# Aura Open Source Developer's Guide

Last updated: July 17, 2014



# Table of Contents

<b>GETTING STARTED.....</b>	<b>1</b>
<b>Chapter 1: Introduction.....</b>	<b>1</b>
Why Use Aura?.....	2
Introduction to Components.....	2
Introduction to Events.....	3
Browser Support.....	3
Aura Version Numbers.....	3
<b>Chapter 2: Quick Start.....</b>	<b>5</b>
Create an Aura App from the Command Line.....	6
Import an Aura App into Eclipse.....	6
Add a Component.....	7
Next Steps.....	8
Build Aura from Source.....	8
Aura Demos.....	9
<b>CREATING COMPONENTS.....</b>	<b>11</b>
<b>Chapter 3: Components.....</b>	<b>11</b>
Component Bundles.....	13
Component IDs.....	13
HTML in Components.....	14
CSS in Components.....	15
Component Attributes.....	16
Component Composition.....	17
Component Body.....	19
Component Facets.....	20
Lazy Loading.....	21
Localization.....	22
Providing Component Documentation .....	23
<b>Chapter 4: Expressions.....</b>	<b>26</b>
Example Aura Expressions.....	28
Value Providers.....	29
Global Value Providers.....	30
Expression Evaluation.....	32
Expression Operators Reference.....	32
Expression Functions Reference.....	35
<b>Chapter 5: User Interface Overview.....</b>	<b>38</b>
Input Components Overview.....	39
Buttons.....	40
Date and Time Fields.....	42

Number Fields.....	43
Text Fields.....	45
Rich Text Fields.....	47
Checkboxes.....	48
Field-level Errors.....	49
Drop-down Lists.....	50
Horizontal Layouts.....	53
Vertical Layouts.....	54
Working with Auto-Complete.....	55
Creating Lists.....	57
<b>Chapter 6: Using Labels.....</b>	<b>60</b>
\$Label.....	61
Input Component Labels.....	61
Dynamically Populating Label Parameters.....	62
Dynamically Creating Labels.....	64
Customizing your Label Implementation.....	65
<b>Chapter 7: Supporting Accessibility.....</b>	<b>68</b>
Accessibility Considerations.....	69
Buttons.....	69
Carousels.....	69
Help and Error Messages.....	69
Forms, Fields, and Labels.....	70
Images.....	70
Using Images.....	71
Events.....	72
<b>COMMUNICATING WITH EVENTS.....</b>	<b>73</b>
<b>Chapter 8: Events.....</b>	<b>73</b>
Handling Events with Client-Side Controllers.....	74
Component Events.....	75
Component Event Example.....	77
Application Events.....	79
Application Event Example.....	80
Event Handling Lifecycle.....	83
Advanced Events Example.....	84
Firing Aura Events from Non-Aura Code.....	88
Events Best Practices.....	88
Events Anti-Patterns.....	89
Events Fired During the Rendering Lifecycle .....	89
<b>CREATING APPS.....</b>	<b>94</b>
<b>Chapter 9: App Basics.....</b>	<b>94</b>

App Overview.....	95
Designing App UI.....	95
Creating App Templates.....	96
<b>Chapter 10: Styling Apps.....</b>	<b>98</b>
Vendor Prefixes.....	98
<b>Chapter 11: Adding Components.....</b>	<b>99</b>
<b>Chapter 12: Using JavaScript.....</b>	<b>100</b>
Accessing the DOM.....	101
Using JavaScript Libraries.....	101
Working with Attribute Values in JavaScript.....	102
Working with a Component Body in JavaScript.....	102
Sharing JavaScript Code in a Component Bundle.....	103
Client-Side Rendering to the DOM.....	105
Client-Side Runtime Binding of Components.....	108
Validating Fields.....	110
Throwing Errors.....	112
JavaScript Services.....	113
<b>Chapter 13: JavaScript Cookbook.....</b>	<b>115</b>
Invoking Actions on Component Initialization.....	116
Detecting Data Changes.....	117
Finding Components by ID.....	117
Dynamically Creating Components.....	118
Dynamically Adding Event Handlers.....	119
Creating a Document-Level Event Handler.....	120
Modifying Components from External JavaScript.....	120
Dynamically Showing or Hiding Markup.....	121
Adding and Removing Styles.....	121
<b>Chapter 14: Using Java.....</b>	<b>123</b>
Essential Terminology.....	124
Reading Initial Component Data with Models.....	124
Java Models.....	124
JSON Models.....	126
Accessing Models in JavaScript.....	127
Creating Server-Side Logic with Controllers.....	128
Creating a Java Server-Side Controller.....	129
Calling a Server-Side Action.....	131
Queueing of Server-Side Actions.....	132
Abortable Actions.....	132
Background Actions.....	133
Caboose Actions.....	134
Storable Actions.....	134

Server-Side Rendering to the DOM.....	136
Server-Side Runtime Binding of Components.....	137
Serializing Exceptions.....	138
<b>Chapter 15: Java Cookbook.....</b>	<b>140</b>
Dynamically Creating Components in Java.....	141
Setting a Component ID.....	141
Getting a Java Reference to a Definition.....	142
<b>Chapter 16: URL-Centric Navigation.....</b>	<b>143</b>
Using Custom Events in URL-Centric Navigation.....	144
Accessing Tokenized Event Attributes.....	144
Using Layouts for Metadata-Driven Navigation.....	144
Using Custom Events in Metadata-Driven Navigation.....	146
<b>Chapter 17: Using Object-Oriented Development.....</b>	<b>147</b>
What is Inherited?.....	148
Inheritance Rules.....	148
Inherited Component Attributes.....	149
Accessing a Super Component.....	150
Traversing a Component's Extension Hierarchy.....	150
Abstract Components.....	151
Interfaces.....	151
Marker Interfaces.....	152
<b>Chapter 18: Caching with Storage Service.....</b>	<b>153</b>
Initializing Storage Service.....	155
Using Storage Service.....	156
<b>Chapter 19: Using the AppCache.....</b>	<b>157</b>
Enabling the AppCache.....	158
Loading Resources with AppCache.....	158
Specifying Additional Resources for Caching.....	158
<b>Chapter 20: Controlling Access.....</b>	<b>160</b>
Application Access Control.....	161
Interface Access Control.....	161
Component Access Control.....	161
Attribute Access Control.....	162
Event Access Control.....	162
<b>TESTING AND DEBUGGING.....</b>	<b>163</b>
<b>Chapter 21: Testing Components.....</b>	<b>163</b>
JavaScript Test Suite Setup.....	164
Assertions.....	166
Debugging Components.....	167

Utility Functions.....	167
Sample Test Cases.....	169
Mocking Java Classes.....	170
Mocking Java Models.....	171
Mocking Java Providers.....	172
Mocking Java Actions.....	173
<b>Chapter 22: Customizing Behavior with Modes.....</b>	<b>174</b>
Modes Reference.....	175
Controlling Available Modes.....	177
Setting the Default Mode.....	177
Setting the Mode for a Request.....	177
<b>Chapter 23: Debugging.....</b>	<b>178</b>
Log Messages.....	179
Warning Messages.....	179
Debugging with Network Traffic.....	179
Aura Debug Tool.....	183
Querying State and Statistics.....	184
<b>CUSTOMIZING AURA.....</b>	<b>186</b>
<b>Chapter 24: Plugging in Custom Code with Adapters.....</b>	<b>186</b>
Default Adapters.....	187
Overriding Default Adapters.....	187
<b>Chapter 25: Accessing Components from Non-Aura Containers.....</b>	<b>189</b>
Add an Aura button inside an HTML div container.....	190
<b>Chapter 26: Customizing Data Type Conversions.....</b>	<b>191</b>
Registering Custom Converters.....	192
Custom Converters.....	193
<b>REFERENCE.....</b>	<b>197</b>
<b>Chapter 27: Reference Overview.....</b>	<b>197</b>
aura:application.....	198
aura:component.....	199
aura:clientLibrary.....	200
aura:dependency.....	202
aura:event.....	203
aura:if.....	203
aura:interface.....	204
aura:iteration.....	205
aura:renderIf.....	206
aura:set.....	208
Setting Attributes on a Super Component.....	208

Setting Attributes on a Component Reference.....	210
Setting Attributes Inherited from an Interface.....	210
Supported HTML Tags.....	211
Supported aura:attribute Types.....	211
Basic Types.....	212
Object Types.....	213
Collection Types.....	214
Custom Java Class Types.....	215
Aura-Specific Types.....	215
<b>APPENDIX.....</b>	<b>218</b>
<b>Chapter 28: Aura Request Lifecycle.....</b>	<b>218</b>
Initial Application Request.....	219
Component Request Lifecycle.....	220
Component Request Overview.....	220
Server-Side Processing for Component Requests.....	221
Client-Side Processing for Component Requests.....	223
Component Request Glossary.....	224
<b>Index.....</b>	<b>227</b>



# GETTING STARTED

## Chapter 1

### Introduction

---

#### In this chapter ...

- [Why Use Aura?](#)
- [Introduction to Components](#)
- [Introduction to Events](#)
- [Browser Support](#)
- [Aura Version Numbers](#)

#### What is Aura?

Aura is a UI framework for developing dynamic web apps for mobile and desktop devices. Aura provides a scalable long-lived lifecycle to support building apps engineered for growth.

Aura supports partitioned multi-tier component development that bridges the client and server. It uses JavaScript on the client side and Java on the server side.



## Why Use Aura?

There are many benefits of using Aura to build apps.

### **Rich Component Set**

Aura comes with a rich and extensible component set to kick start building apps. You don't have to spend your time optimizing your apps for different devices as the components take care of that for you.

### **Performance**

Aura uses a stateful client and stateless server architecture that relies on JavaScript on the client-side to manage UI component metadata and application data. Aura uses JSON to exchange data between the server and the client. To maximize efficiency, the server only sends data that is needed by the user.

The framework intelligently utilizes your server, browser, devices, and network so you can focus on the logic and interactions of your apps.

### **Event-driven architecture**

Aura uses an event-driven architecture for better decoupling between components. Any component can subscribe to an application event, or to a component event they can see.

### **Model-View-Controller (MVC) architecture**

Aura markup represents the view and defines the component's public shape via attributes and events. Components have models and controllers to represent the data set and logic respectively.

### **Parallel design and development**

Building an app with components facilitates parallel design, improving overall development efficiency. Aura provides the basic constructs of inheritance, polymorphism, and encapsulation from object-oriented programming and applies them to presentation layer development. With Aura, you can extend a component or implement a component interface. 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.

### **Device-aware and cross browser compatibility**

Apps built on the Aura framework are responsive and provide an enjoyable user experience. Aura supports the latest in browser technology such as HTML5, CSS3, and touch events.

## Introduction to Components

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

The framework includes a rich set of prebuilt 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:

[Components](#)

## Introduction to 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.

An Aura component registers that it may fire an event in its markup. Aura 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 Aura:

- **Component events** are handled by the component itself or a component that instantiates or contains the component.
- **Application events** are essentially a traditional publish-subscribe model. All components that provide a handler for the event are notified when the event is fired.

You write the handlers in JavaScript controller actions.

### See Also:

[Events](#)

[Handling Events with Client-Side Controllers](#)

## Browser Support

Aura supports the most recent stable version of the following web browsers across major platforms, with exceptions noted.

Browser	Notes
Google Chrome™	
Apple® Safari® 5+	For Mac OS X and iOS
Mozilla® Firefox®	
Microsoft® Internet Explorer®	We recommend using Internet Explorer 9, 10, or 11. Internet Explorer 7 and 8 may provide a degraded performance.



**Note:** For all browsers, you must enable JavaScript. We recommend enabling cookies.

## Aura Version Numbers

Aura uses version numbers that are consistent with other Maven projects. This makes it easy for projects built with Maven to express their dependency on Aura.

The version number scheme is:

**major.minor[.incremental] [-qualifier]**

The `major`, `minor`, and optional `incremental` parts are all numeric. The `qualifier` string is optional. For example, `1.2.0`, `2.4`, or `2.5.0-SNAPSHOT` are all valid.

The `major` number advances and the `minor` and `incremental` counters reset to zero for releases with large functional changes. Within a `major` release, the `minor` number advances for small updates with enhancements and bug fixes. The `incremental` counter is only used for targeted fixes, usually for critical bugs.

The `qualifier` string is largely arbitrary. A version number that includes a `qualifier` is a non-release build. The compatibility guarantee is weaker, because the build is stabilizing towards a release. In order of increasing stability, the `qualifier` may be:

#### **SNAPSHOT**

An arbitrary development build. There are no assurances for such a build, as its under active development.

#### **msN**

A milestone build. Some features can at least be demonstrated, but the build isn't ready for a full release. Feature behavior may change as the milestone progresses towards a release.

#### **rcN**

A release candidate, which is a build we think is close to a final release. However, it's still undergoing final checking and may change before an unqualified release.

A release build has a fixed `major`, `minor`, and `incremental` version. It's newer and preferable to any unqualified version with the same version number. For example, `x.y.z` is newer than `x.y.z-SNAPSHOT`.

Release candidates are always newer than any milestone, and a release candidate or milestone with a higher number is newer than others with lower numbers.

If you have the source code for the Aura framework, you can find the version number in the root folder's `pom.xml` file. For example:

```
<project ... >
  <name>Aura Framework</name>
  <version>0.273</version>
</project>
```

Although it will rarely be important, you can use the `Java ConfigAdapter.getAuraVersion()` method to see what version of Aura is running your code.

# Chapter 2

## Quick Start

---

### In this chapter ...

- [Create an Aura App from the Command Line](#)
- [Import an Aura App into Eclipse](#)
- [Next Steps](#)

The quick start steps you through building and running your first Aura app from the command line, or in the Eclipse IDE. Choose the method you're most comfortable with and check out the next steps after you build an app.

## Create an Aura App from the Command Line

You can generate a basic Aura app quickly using the command line. For details, see the `README.md` file in the [Aura repo](#).

### See Also:

[Import an Aura App into Eclipse](#)

[Next Steps](#)

## Import an Aura App into Eclipse

This section shows you how to import the Aura app you created in the command-line quick start into Eclipse.



**Note:** You must complete the [command-line quick start](#) before proceeding.

Before you begin, make sure you have this software installed:

1. [JDK 1.7](#)
2. [Apache Maven 3](#)
3. [Eclipse 3.7 or later](#) and the [m2eclipse plugin](#). Choose the Eclipse distribution for Java EE Developers. This includes JavaScript editing and other Web UI tools.

### Step 1: Import the Command-Line Project into Eclipse

1. Click **File** > **Import...** > **Maven** > **Existing Maven Projects**.
2. Click **Next**.
3. In the **Root Directory** field, browse to the `helloWorld` folder created in the command-line quick start and click **OK**.
4. Click **Finish**.

You should now have a new project called `helloWorld` in the Package Explorer.

### Step 2: Build and Run Your Project

1. Click **Run** > **Debug Configurations....**
2. Double click **Maven Build**.
3. Enter these values:
  - **Name:** HelloWorld Server
  - **Base directory:** `${workspace_loc:/helloWorld}` (where `helloWorld` is the same as your Artifact Id)
  - **Goals:** `jetty:run`



**Note:** To use another port, such as port 8080, append `-Djetty.port=8080` to `jetty:run`.

4. Click **Debug**.

You should see a message in the Eclipse Console window indicating that the Jetty server has started.

### Step 3: Test Your App

1. Navigate to `http://localhost:8080` to test your app.

You will be redirected to `http://localhost:8080/helloWorld/helloWorld.app`.

2. Validate that your app is working by looking for "hello web" in the browser page.

### See Also:

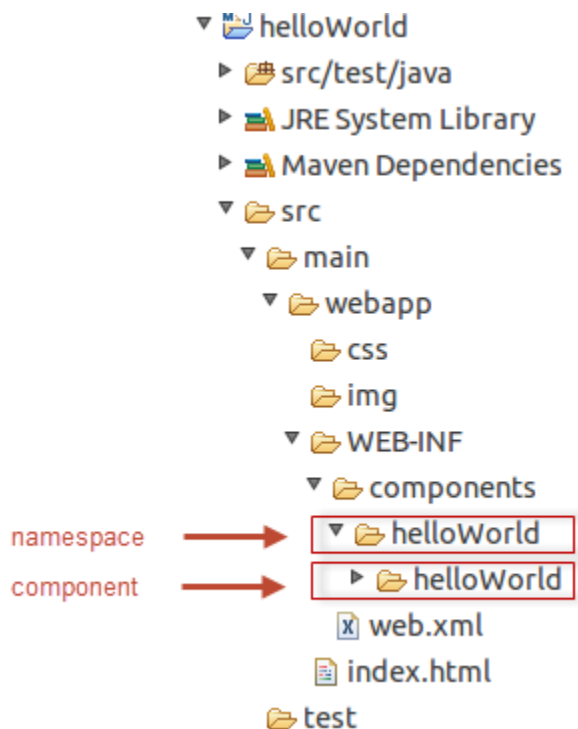
[Browser Support](#)

## Add a Component

An Aura app is represented by a `.app` file composed of Aura components and HTML tags.

Components are the building blocks in your app and are grouped in a namespace. In addition to the required top-level `<aura:component>` tag in a component or `<aura:application>` tag in an application, you can insert user interface components using tags defined in the Aura component library.

In Eclipse, we'll add a component to our simple app. The following diagram shows the folder structure for the project. Under the `components` folder, there is a `helloWorld` folder representing the namespace. Under that folder is a sub-folder, also called `helloWorld`, which represents the application, which is a special type of component. This folder can also contain resources, such as CSS and JavaScript files. We will add a new component to the `helloWorld` namespace.



### Step 1: Make a New Component

1. In Eclipse Package Explorer, right-click the `helloWorld` namespace folder under `components` and select **New > File**.
2. Create a new `hello` component in the namespace by entering these values:

**Parent folder:** `helloWorld/src/main/webapp/WEB-INF/components/helloWorld/hello`

**File name:** `hello.cmp`



**Note:** We're adding the component to a new `hello` folder under the `helloWorld` namespace folder.

3. Click **Finish**.

4. Open `hello.cmp` and enter:

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

5. Save the file.
6. View the component in a browser by navigating to `http://localhost:8080/helloWorld/hello.cmp`. If the component is not displayed, make sure that the web server is running.

## Step 2: Add the Component to the App

Now, we're going to add our new component to the app. In this case, the component is simple, but the intent is to demonstrate how you can create a component that is reusable in multiple apps.

1. Open `helloWorld.app` and replace its contents with:

```
<aura:application>

    <h1>My First Aura App</h1>
    <helloWorld:hello />

</aura:application>
```

2. Save the file.
3. View the app in a browser by navigating to `http://localhost:8080/helloWorld/helloWorld.app`.

You created an app and added a simple component using Eclipse. Aura enables you to use JavaScript on the client and Java on the server to create rich applications, as you'll see in later topics.

## See Also:

[aura:application](#)  
[Component Body](#)

## Next Steps

Now that you've created your first app, you might be wondering where do I go from here? There is much more to learn about Aura. Here are a few ideas for next steps.

- [Look at the Aura source code and build it from source in Eclipse](#)
- [Explore the capabilities of the Aura framework through the Aura Note sample app.](#)
- [Browse components that come out-of-the-box with Aura.](#)

## Build Aura from Source

You don't have to build Aura from source to use it. However, if you want to customize the source code or submit a pull request with enhancements to the framework, here's how to do it. Before you begin, make sure you have this software installed:

1. [JDK 1.7](#)
2. [Apache Maven 3](#)

### Step 1: Install git



The Aura source code is available on GitHub. To download the source code, you need an account on GitHub and the `git` command-line tool.

1. Create a GitHub account at <https://github.com/signup/free>.
2. Follow the instructions at <https://help.github.com/articles/set-up-git> to install and configure `git` and `ssh` keys.

You don't have to create your own repository. You'll be cloning the Aura source next.

### Step 2: Get and Build Aura Source

1. On the command line, navigate to the directory where you want to keep the Aura source code.
2. Run the following commands to clone the source with `git` and build it with Maven:

```
git clone git@github.com:forcedotcom/aura.git
cd aura
mvn install
```

You should see a message that the build completed successfully.

### Step 3: Import Aura Source into Eclipse

You can use your IDE of choice. These instructions show you how to import the Aura source into Eclipse.

1. Install [Eclipse 3.7 or later](#) and the [m2eclipse plugin](#). Choose the Eclipse distribution for Java EE Developers. This includes JavaScript editing and other Web UI tools..
2. Import the Aura source by clicking **File > Import > Maven > Existing Maven Projects**.
3. Click **Next**.
4. In the **Root Directory** field, browse to the directory that you cloned.
5. Click **Next**.
6. Click **Finish**.

You should see the source in the Package Explorer.

### Step 4: Run Aura from Eclipse

To run Aura's Jetty server from Eclipse:

1. Click **Window > Preferences > Maven > Installations > Add...**
2. Navigate to your Maven installation and select it.
3. Click **Run > Debug Configurations...**
4. Right click **Maven Build** and select **New**.
5. Enter `Aura Jetty` in the **Name** field.
6. In the **Base directory** field, click **Browse Workspace...**
7. Select `aura-jetty` and click **OK**.
8. Enter `jetty:run` in the **Goals** field.
9. Click **ApplyApply**.
10. Click **Debug**.

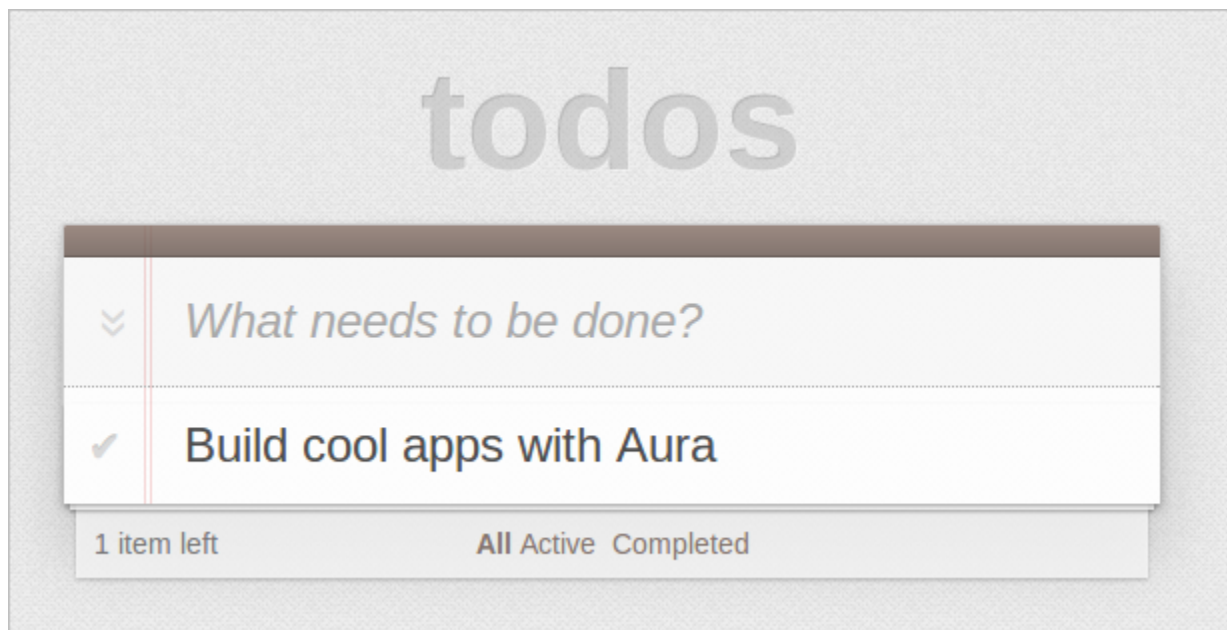
In the Console window, you should see a message that the Jetty server started. In a browser, navigate to `http://localhost:9090/` to access the server.

## Aura Demos

### TodoMVC

The TodoMVC app demonstrates the core concepts of the Aura framework.

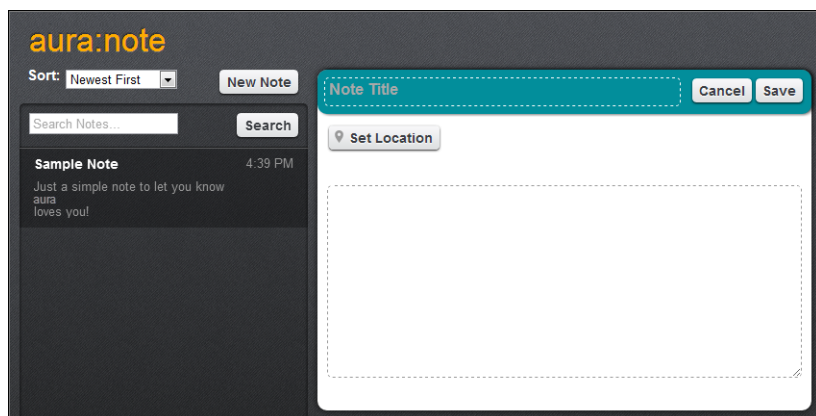
[Get the source](#)



### Aura Note

Aura Note is a note-taking app showcasing the simplicity of building apps on Aura.

[Get the source](#)



# CREATING COMPONENTS

## Chapter 3

### Components

#### In this chapter ...

- [Component Bundles](#)
- [Component IDs](#)
- [HTML in Components](#)
- [CSS in Components](#)
- [Component Attributes](#)
- [Component Composition](#)
- [Component Body](#)
- [Component Facets](#)
- [Lazy Loading](#)
- [Localization](#)
- [Providing Component Documentation](#)

Components are the functional units of Aura. They encapsulate a modular and potentially reusable section of UI, and can range in granularity from a single line of text to an entire application.

#### Creating a Component

Component files 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 Aura component definition.

Aura 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 Aura components here-->
    </div>
</aura:component>
```



**Note:** Everything in Aura markup is case insensitive except for references to JavaScript, CSS, or Java.

#### Component Namespace

Every component is part of a namespace, which is used to group related components together. Another component or application can reference a component by adding `<myNamespace:myComponent>` in its markup. For example, the `helloWorld` component is in the `auradocs` namespace. Another

component can reference it by adding `<auradocs:helloWorld />` in its markup.

Note where this component file is stored in the filesystem:

```
aura-components/components/auradocs/helloWorld/helloWorld.cmp
```

All core Aura components are in the `aura-components/components` directory. All folders within that directory map to a namespace. For example, all the components related to this documentation live in the `auradocs` namespace.

Each folder within a namespace folder maps to a specific component and contains all the resources necessary for the component. We refer to this folder as the component's bundle.

In this case, the `helloWorld` bundle only contains a `helloWorld.cmp` file, which has the markup for this component. See [Component Bundles](#) for more information on files you can include in the bundle.

## Viewing Components

So we've got a component, but the point of Aura is to build web applications. How do we actually view this component in a web browser?

In DEV mode, you can address any Aura component using the URL scheme `http://<myServer>/<namespace>/<component>.cmp`



**Note:** DEV mode is the default mode when you're developing. When Aura is running in PROD mode, you can't directly address an Aura component using a `.cmp` suffix. However, you can directly address an application with a `.app` suffix. For more information, see [Modes](#).

Open up a new tab in your browser and try it yourself. Navigate to `/auradocs/helloWorld.cmp`. Or better yet, create your own practice namespace in your filesystem and follow along, creating your own versions of the sample components. Either way, at the end you should see a heart-warming "Hello, world!" in your browser.

## Support Level

Each component has a support level ranging from fully supported (GA) to new and experimental (PROTO). The support level is defined in the `support system` attribute in the `<aura:component>` tag. For more information, see the [Reference tab](#).

### See Also:

[Component Bundles](#)

[Customizing Behavior with Modes](#)

[Modes Reference](#)

## Component Bundles

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

File	File Name	Usage	See Also
Component or Application	<code>sample.cmp</code> or <code>sample.app</code>	The only required resource in a bundle. Contains markup for the component or app. Each bundle contains only one component or app resource.	<a href="#">Components</a> <a href="#">aura:application</a>
CSS Styles	<code>sample.css</code>	Styles for the component.	<a href="#">CSS in Components</a>
Controller	<code>sampleController.js</code>	Client-side controller methods to handle events in the component.	<a href="#">Handling Events with Client-Side Controllers</a>
Documentation	<code>sample.auradoc</code>	A description, sample code, and one or multiple references to example components	<a href="#">Providing Component Documentation</a>
Model	<code>sampleModel.js</code>	JSON model to initialize a component.	<a href="#">JSON Models</a>
Renderer	<code>sampleRenderer.js</code>	Client-side renderer to override default rendering for a component.	<a href="#">Client-Side Rendering to the DOM</a>
Helper	<code>sampleHelper.js</code>	Helper methods that are shared by the controller and renderer.	<a href="#">Sharing JavaScript Code in a Component Bundle</a>
Provider	<code>sampleProvider.js</code>	Client-side provider that returns the concrete component to use at runtime.	<a href="#">Client-Side Runtime Binding of Components</a>
Test Cases	<code>sampleTest.js</code>	Contains a test suite to be run in the browser.	<a href="#">Testing Components</a>

All resources in the component bundle are auto-wired. For example, a controller `<componentName>Controller.js` is auto-wired to its component.

## Component IDs

A component has two types of IDs: a local ID and a global ID.

### Local IDs

A local ID is unique within a component and is only scoped to the component.

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

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

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

To retrieve a component by local ID in JavaScript code, use the `component.find("localId")` method. For more information, see the [JavaScript API](#).

## Global IDs

Every component has a unique `globalId`, which is the generated runtime-unique ID of the component instance. A global ID is not guaranteed to be the same beyond the lifetime of a component, so it should never be relied on for tests.

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>
```

You can use the `getGlobalId()` function in JavaScript to get a component's global ID.

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

You can also do the reverse operation and get a component if you have its global ID.

```
var comp = $A.getCmp(globalId);
```

## See Also:

[Finding Components by ID](#)

# HTML in Components

An HTML tag is treated as a first-class component in Aura. Each HTML tag is translated into an Aura component, allowing it to enjoy the same rights and privileges as any other component.

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. For a full list of unsupported tags, see [Supported HTML Tags](#).

## 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 a model or controller, such as in `<aura:unescapedHtml value="{!m.htmlOutput}"/>` and `<aura:unescapedHtml value="{!v.note.body}"/>`.

`{!<expression>}` is Aura's expression syntax. For more information, see [Expressions](#) on page 26.

## See Also:

[Supported HTML Tags](#)

[CSS in Components](#)

## CSS in Components

Style your components with CSS.

To add CSS to a component, add a new file to the component bundle called `<componentName>.css`. Aura automatically picks up this new file and auto-wires it when the component is used in a page.

For external CSS resources, see [Styling Apps](#) on page 98.

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 blowing away another component's styling. Aura 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



The top-level elements match the `THIS` class and render with a grey background.

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 Java. 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 in a component's markup to add an attribute to the component. Let's look at a sample component, `helloAttributes.cmp`:

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

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 names must start with a letter or underscore. They can also contain numbers or hyphens after the first character.



**Note:** You can't use attributes with hyphens in expressions. For example, `cmp.get("v.name-withHyphen")` is supported, but not `<ui:button label="{!v.name-withHyphen}" />`.

If you load `helloAttributes.cmp` in your browser, it doesn't look any different from the `helloWorld.cmp` component that we looked at earlier.

Now, append `?whom=you` to the URL and reload the page. The value in the query string sets the value of the `whom` attribute. Supplying attribute values via the query string when requesting a component is one way to set the attributes on that component.



**Warning:** This only works for attributes of type `String`.

### Expressions

In the markup for `helloAttributes.cmp` you'll see a line `Hello {!v.whom}!`. This is what's responsible for the component's dynamic output.



{!<expression>} is Aura'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.` (for "view", as in "model-view-controller") is the value provider for a component's attribute set. We'll see examples of the model and controller value providers later.

## Attribute Validation

We defined the set of valid attributes in `helloAttributes.cmp`, so Aura will automatically validate that only valid attributes are passed to that component.

Try requesting `helloAttributes.cmp` with the query string `?fakeAttribute=fakeValue`. You should receive an error that `helloAttributes.cmp` doesn't have a `fakeAttribute` attribute.

## See Also:

[Supported aura:attribute Types](#)  
[Expressions](#)

# 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.

`nestedComponents.cmp` shows an example of including components inside other components.

## Component source

```
<aura:component>
  Observe! Components within components!

  <auradocs:helloHTML/>

  <auradocs: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: we just reference the component by its "descriptor", which is of the form `<namespace>:<component>`. `nestedComponents.cmp` references the `helloHTML.cmp` component, which lives in the `auradocs` namespace. Hence, its descriptor is `auradocs:helloHTML`.

Note how `nestedComponents.cmp` also references `auradocs:helloAttributes`. Just like adding attributes to an HTML tag, you can specify attribute values to an Aura component as part of the component tag. `nestedComponents.cmp` sets the `whom` attribute of `helloAttributes.cmp` to "component composition", which affects the output of the component.

Here is the source for `helloHTML.cmp`.

## 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



Here is the source for `helloAttributes.cmp`.

### Component source

```

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

```

### 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 `auradocs:helloAttributes`.

### Component source

```

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

```

```

    <auradocs:helloHTML/>

    <auradocs: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!

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

## Definitions versus Instances

If you're familiar with object-oriented programming, you know the difference between a class and an instance of that class. Aura works the same way: when you create a `.cmp` file, you are providing the definition (class) of that component. When you put a component tag in a `.cmp` file, 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 `auradocs:helloAttributes` with a different attribute value. The two instances of the `auradocs:helloAttributes` component have different values for their `whom` attribute.

## Component source

```

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

    <auradocs:helloHTML/>

    <auradocs:helloAttributes whom="{!v.passthrough}"/>

    <auradocs: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 `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 value of an inherited attribute, use the `<aura:set>` tag.

There are only a few tags that are allowed inside `<aura:component>`. These include but are not limited to `<aura:attribute>`, `<aura:registerEvent>`, `<aura:handler>`, and `<aura:set>`. Any free markup that is not enclosed in one of the tags allowed in a component is assumed to be part of the body. It's 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">`.

```
<aura:component>
  <div>Body part</div>
  <ui:button label="Push Me"/>
</aura:component>
```

This is a shortcut for:

```
<aura:component>
  <aura:set attribute="body">
    <div>Body part</div>
    <ui:button label="Push Me"/>
  </aura:set>
</aura:component>
```

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

```
<ui:panel>
  Hello world!
</ui:panel>
```

This is a shortcut for:

```
<ui:panel>
  <aura:set attribute="body">
    Hello World!
  </aura:set>
</ui:panel>
```

## 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[]`.

The `body` attribute is an example of a facet. The only difference between facets that you define and `v.body` is that the shorthand of optionally omitting the `aura:set` tag only works for `v.body`.

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

### Component source

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

  <div>
    <span class="header">{!v.header}</span><br/>
    <span class="body">{!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. The following component, `helloFacets.cmp`, sets these attributes.

### Component source

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

  <auradocs:facetHeader>

    Nice body!

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

</aura:component>
```

### See Also:

[Component Body](#)

## Lazy Loading

Lazy loading a component can improve the apparent response time of your app if you have many components that users don't need all at once.

A lazily loaded component is rendered after its parent component is loaded. When a component is lazily loaded, the app renders a placeholder spinner component, `aura:placeholder`. The framework makes an asynchronous request and replaces the placeholder component once the request returns the component.

To enable lazy loading on a component, add the `aura:load="lazy"` attribute to your component tag.



**Note:** When you enable lazy loading on a component, you can't pass a body or other non-primitive attribute values into the component.

### Loading a Component Exclusively

To load the component in its own request, add the `aura:load="exclusive"` attribute to your component tag. This is similar to lazy loading except that the request to get the component won't be grouped with any other actions or the retrieval

of other lazy-loaded components. The component will have its own request so that it doesn't block the loading of anything else.

### See Also:

[Supported aura:attribute Types](#)

## Localization

Aura provides client-side localization support on input and output components.

The components retrieve the browser's locale information and display the date and time accordingly. The following example shows how you can override the default `langLocale` and `timezone` attributes. The output displays the time in the format `hh:mm` by default.

### Component source

```
<aura:component>
  <ui:outputDateTime value="2013-05-07T00:17:08.997Z" timezone="Europe/Berlin"
    langLocale="de"/>
</aura:component>
```

The component renders as `Mai 7, 2013 2:17:08 AM`.

Additionally, you can use the global value provider, `$Locale`, to obtain a browser's locale information. By default, Aura uses the browser's locale, but it can be configured to use others through the global value provider.

### Using the Localization Service

Aura's localization service enables you to manage the localization of date, time, numbers, and currencies. For available methods, see [AuraLocalizationService](#) in the JavaScript API.

This example sets the formatted date time using `$Locale` and the localization service.

```
var dateFormat = $A.getGlobalValueProviders().get("$Locale.dateFormat");
var dateString = $A.localizationService.formatDateTime(new Date(), dateFormat);
```

If you're not retrieving the browser's date information, you can specify the date format on your own. This example specifies the date format and uses the browser's language locale information.

```
var dateFormat = "MMMM d, yyyy h:mm a";
var userLocaleLang = $A.get("$Locale.langLocale");
return $A.localizationService.formatDate(date, dateFormat, userLocaleLang);
```

This example compares two dates to check that one is later than the other.

```
if( $A.localizationService.isAfter(StartDateTime,EndDateTime)) {
  //throw an error if StartDateTime is after EndDateTime
}
```

### See Also:

[Global Value Providers](#)

## Providing Component Documentation

Component documentation helps others understand and use your components.

You can provide two types of component reference documentation:

- Documentation definition (DocDef): Full documentation on a component, including a description, sample code, and a reference to an example. DocDef supports extensive HTML markup and is useful for describing what a component is and what it does.
- Inline descriptions: Text-only descriptions, typically one or two sentences, set via the `description` attribute on the `Aura` tag.

To provide a DocDef, create a `.auradoc` file in the component bundle and use the `<aura:documentation>` tag to wrap your documentation. The following example shows the documentation definition (DocDef) for the `ui:button` component.



**Note:** DocDef is currently supported for components and applications. Events and interfaces support inline descriptions only.

```
<aura:documentation>
  <aura:description>
    <p>
      A <code>ui:button</code> component represents a button element that executes an action
      defined by a controller.
      Clicking the button triggers the client-side controller method set for the
      <code>press</code> event.
      The button can be created in several ways.
    </p>
    <p>
      A text-only button has only the required <code>label</code> attribute set on it.
      To create a button with both image and text, use the <code>label</code> attribute and
      add styles for the button.
    </p>
    <p>The visual appearance of buttons is highly configurable, as are text and accessibility
    attributes.</p>

    <!--More markup here, such as <pre> for code samples-->
    <p>The markup for a button with text and image results in the following HTML. </p>
    <pre>
      <button class="default uiBlock uiButton" accesskey type="button">
        
        <span class="label bBody truncate" dir="ltr">Find</span>
      </button>
    </pre>

  </aura:description>
  <aura:example name="buttonExample" ref="uiExamples:buttonExample" label="Using ui:button">

    <p>This example shows a button that displays the input value you enter.</p>
  </aura:example>
  <aura:example name="buttonSecondExample" ref="uiExamples:buttonSecondExample"
  label="Customizing ui:button">
    <p>This example shows a customized <code>ui:button</code> component.</p>
  </aura:example>
</aura:documentation>
```

A documentation definition contains these tags.

Tag	Description
<code>&lt;aura:documentation&gt;</code>	The top-level definition of the DocDef

Tag	Description
<code>&lt;aura:description&gt;</code>	Describes the component using extensive HTML markup. To include code samples in the description, use the <code>&lt;pre&gt;</code> tag, which renders as a code block. Code entered in the <code>&lt;pre&gt;</code> tag must be escaped. For example, escape <code>&lt;aura:component&gt;</code> by entering <code>&amp;lt;aura:component&amp;gt;</code> .
<code>&lt;aura:example&gt;</code>	References an example that demonstrates how the component is used. Supports extensive HTML markup, which displays as text preceding the visual output and example component source. The example is displayed as interactive output. Multiple examples are supported and should be wrapped in individual <code>&lt;aura:example&gt;</code> tags. <ul style="list-style-type: none"> <li><code>name</code>: The API name of the example</li> <li><code>ref</code>: The reference to the example component in the format <code>&lt;namespace:exampleComponent&gt;</code></li> <li><code>label</code>: The label of the title</li> </ul>

## Providing an Example Component

Recall that the DocDef includes a reference to an example component. The example component is rendered as an interactive demo in the component reference documentation when it's wired up using `aura:example`.

```
<aura:example name="buttonExample" ref="uiExamples:buttonExample" label="Using ui:button">
```

The following is an example component that demonstrates how `ui:button` can be used.

```
<!--The uiExamples:buttonExample example component -->
<aura:component>
    <ui:inputText aura:id="name" label="Enter Name:" placeholder="Your Name" />
    <ui:button aura:id="button" buttonTitle="Click to see what you put into the field"
        class="button" label="Click me" press="{!c.getInput}" />
    <ui:outputText aura:id="outName" value="" class="text" />
</aura:component>
```

## Providing Inline Descriptions

Inline descriptions provide a brief overview of what an element is about. HTML markup is not supported in inline descriptions. These tags support inline descriptions via the `description` attribute.

Tag	Example
<code>&lt;aura:component&gt;</code>	<code>&lt;aura:component description="Represents a button element"&gt;</code>
<code>&lt;aura:attribute&gt;</code>	<code>&lt;aura:attribute name="langLocale" type="String" description="The language locale used to format date value."/&gt;</code>
<code>&lt;aura:event&gt;</code>	<code>&lt;aura:event type="COMPONENT" description="Indicates that a keyboard key has been pressed and released"/&gt;</code>



Tag	Example
<code>&lt;aura:interface&gt;</code>	<code>&lt;aura:interface description="A common interface for date components"/&gt;</code>
<code>&lt;aura:registerEvent&gt;</code>	<code>&lt;aura:registerEvent name="keydown" type="ui:keydown" description="Indicates that a key is pressed"/&gt;</code>

**See Also:**[Reference Overview](#)

# Chapter 4

## Expressions

### In this chapter ...

- [Example Aura Expressions](#)
- [Value Providers](#)
- [Expression Evaluation](#)
- [Expression Operators Reference](#)
- [Expression Functions Reference](#)

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

An Aura 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 in Aura 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.

The resulting value can be a primitive (integer, string, and so on), a boolean, a JavaScript or Aura object, an Aura component or collection, a controller method such as an action method, and other useful results.



### Important:

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 Aura, `{ ! }` is simply the delimiter used to begin an expression in Aura.

Identifiers in an expression, such as attribute names accessed through the view, model values, 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.

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.

**See Also:**

*[Example Aura Expressions](#)*

## Example Aura Expressions

Here are a few examples of Aura expressions that illustrate different types of usage.

### Dynamic Output

The simplest way to use expressions is to simply output them. Values used in the expression can be from component attributes, or property values on a model object retrieved from a server, literal values, and so on.

```
<p>{!v.desc}</p>
```

In the expression `{!v.desc}`, `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.

```
<div>
  <h2>{! 'Title: ' + m.title}</h2>
  <p class="note-body">{!m.body}</p>
</div>
```

The expressions here display values retrieved from the server, held in an object `m`, the component's model. The title attribute is concatenated with the literal value `"Title: "`.

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.

### Passing Values

Use expressions to pass values around in Aura. For example:

```
<ui:button label="{!m.myLabel}"/>
```

The `{!m.myLabel}` expression passes the `myLabel` field from the model to the `ui:button` component. The expression is not evaluated yet. When the `ui:button` component renders, it evaluates the expression to retrieve the label.

```
<ui:button aura:id="newNote" label="New Note" press="{!c.createNote}"/>
```

The expression `{!c.createNote}` is used to assign a controller action to the `press` attribute of a button component. `c` represents the controller for the component, and `createNote` is the action.

### Conditional Expressions

Although conditional expressions are really just a special case of the previous two, it's worth seeing a few examples.

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

The expression `{!v.location == '/active' ? 'selected' : ''}` is used to conditionally set 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`.

```
<aura:if isTrue="{!m.userHasPermission}">
  <ui:button label="Edit"/>
```

```

    <aura:set attribute="else">
        You don't have permission to edit this.
    </aura:set>
</aura:if>

```

This snippet uses the `<aura:if>` component to conditionally display an edit button if you have a permission that is tracked in the component's model.

### See Also:

[Value Providers](#)

[Handling Events with Client-Side Controllers](#)

[Reading Initial Component Data with Models](#)

## 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 most common value providers are `m`, `v`, and `c` as in "model-view-controller".

Value Provider	Description
<code>m</code>	A component's model with data persisted on a back end service
<code>v</code>	A component's attribute set
<code>c</code>	A component's controller with actions and event handlers for the component

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

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` or `m.title`.



**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.

### Accessing Fields and Related Objects

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

For example, if a component has an attribute `note`, access a note value such as `title` using the `v.note.title` syntax. This example shows usage of this nested syntax for a few attributes.

```

<aura:component>
    <aura:attribute name="note" type="java://org.auraframework.demo.notes.Note"/>
    <ui:block>
        <aura:set attribute="right">
            <ui:outputDateTime value="{!v.note.createdOn}" format="h:mm a"/>
        </aura:set>
    </ui:block>
    {!v.note.title}
</aura:component>

```

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

### See Also:

[Example Aura Expressions](#)

## Global Value Providers

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

The global value providers are:

- `$Label`—See [\\$Label](#) on page 61.
- `globalID`—See [Component IDs](#) on page 13.
- `$Browser`—See [\\$Browser](#) on page 30.
- `$Locale`—See [\\$Locale](#) on page 31.

### `$Browser`

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

Attribute	Description
<code>formFactor</code>	Returns a <code>FormFactor</code> enum value based on the type of hardware the browser is running on. <ul style="list-style-type: none"> <li>• <code>DESKTOP</code> for a desktop client</li> <li>• <code>PHONE</code> for a phone including a mobile phone with a browser and a smartphone</li> <li>• <code>TABLET</code> for a tablet client (for which <code>isTablet</code> returns <code>true</code>)</li> </ul>
<code>isAndroid</code>	Indicates whether the browser is running on an Android device ( <code>true</code> ) or not ( <code>false</code> ).
<code>isIOS</code>	Not available in all implementations. Indicates whether the browser is running on an iOS device ( <code>true</code> ) or not ( <code>false</code> ).
<code>isIPad</code>	Not available in all implementations. Indicates whether the browser is running on an iPad ( <code>true</code> ) or not ( <code>false</code> ).
<code>isIPhone</code>	Not available in all implementations. Indicates whether the browser is running on an iPhone ( <code>true</code> ) or not ( <code>false</code> ).
<code>isPhone</code>	Indicates whether the browser is running on a phone including a mobile phone with a browser and a smartphone ( <code>true</code> ), or not ( <code>false</code> ).
<code>isTablet</code>	Indicates whether the browser is running on an iPad or a tablet with Android 2.2 or later ( <code>true</code> ) or not ( <code>false</code> ).
<code>isWindowsPhone</code>	Indicates whether the browser is running on a Windows phone ( <code>true</code> ) or not ( <code>false</code> ). Note that this only detects Windows phones and does not detect tablets or other touch-enabled Windows 8 devices.

This example shows how to get some `$Browser` attributes.

#### Component source

```
<aura:component>
<pre>
```

```

[isTablet={!$Browser.isTablet}]
[isPhone={!$Browser.isPhone}]
[isAndroid={!$Browser.isAndroid}]
[formFactor={!$Browser.formFactor}]
</pre>
</aura:component>

```

## \$Locale

The \$Locale global value provider gives you information about the browser's locale.

Attribute	Description	Sample Value
language	Returns the language code.	"en", "de", "zh"
country	Returns the ISO 3166 representation of the country code.	"US", "DE", "GB"
variant	Returns the vendor and browser specific code.	"WIN", "MAC", "POSIX"
timezone	Returns the time zone ID based on Java's <code>java.util.TimeZone</code> package.	"EST", "PST", "GMT", "America/New_York"
numberformat	Returns the number formatting based on Java's <code>DecimalFormat</code> class.	"#,##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) replace # to represent trailing zeros.
decimal	Returns the decimal separator.	","
grouping	Returns the grouping separator.	","
percentformat	Returns the percent formatting.	"#,##0%"
currencyformat	Returns the currency formatting.	"¤#,##0.00;(¤#,##0.00)" ¤ represents the currency sign, which is replaced by the currency symbol.
currency_code	Returns the ISO 4217 representation of the currency code.	"USD"
currency	Returns the currency symbol.	"\$"

This example shows how to get some \$Locale attributes.

### Component source

```

<aura:component>
<pre>
[language={!$Locale.language}]
[timezone={!$Locale.timezone}]
[numberformat={!$Locale.numberFormat}]
[currencyformat={!$Locale.currencyFormat}]
</pre>
</aura:component>

```

Aura also provides localization support for input and output components.

### See Also:

[Localization](#)

## 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 simplistic answer is, as often as they need to be. A more complete answer is that Aura can notice when things change, and trigger re-rendering of any components that are affected. Dependencies are handled automatically. This is one of the fundamental benefits of the Aura framework. It knows when to re-render something on the page. 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". Some components simplify assigning actions to user interface events using other attributes, such as the `press` attribute on `<ui:button>`.

Action methods must be assigned to attributes using an expression, for example `{!c.theAction}`. This assigns an `Aura.Action`, which is 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:

```
<ui:button aura:id="likeBtn"
  label="{!(m.likeId == null) ? 'Like It' : 'Unlike It'}"
  press="{!(m.likeId == null) ? c.likeIt : c.unlikeIt}"
/>
```

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.

## Expression Operators Reference

The Aura 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.



Operator	Usage	Description
<code>%</code>	<code>5 % 2</code>	Return the integer remainder of dividing the first number by the second.
<code>-</code>	<code>-m.exp</code>	Unary operator. Reverses the sign of the succeeding number. For example if the value of <code>expenses</code> is 100, then <code>-expenses</code> is -100.

## Numeric Literals

Literal	Usage	Description
Integer	<code>2</code>	Integers are numbers without a decimal point or exponent.
Float	<code>3.14</code> <code>-1.1e10</code>	Numbers with a decimal point, or numbers with an exponent.
Null	<code>null</code>	A literal null number. Matches the explicit null value <b>and</b> numbers with an undefined value.

## String Operators

Expressions based on string operators result in string values.

Operator	Usage	Description
<code>+</code>	<code>'Title: ' + m.note.title</code>	Concatenates two strings together.

## String Literals

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

Literal	Usage	Description
string	<code>'hello world'</code>	Literal strings must be enclosed in single quotation marks. Double quotation marks are reserved for enclosing Aura attributes, and must be escaped in strings.
<code>\&lt;escape&gt;</code>	<code>'\n'</code>	<p>Whitespace characters:</p> <ul style="list-style-type: none"> <li><code>\t</code> (tab)</li> <li><code>\n</code> (newline)</li> <li><code>\r</code> (carriage return)</li> </ul> <p>Escaped characters:</p> <ul style="list-style-type: none"> <li><code>\"</code> (literal <code>"</code>)</li> <li><code>\'</code> (literal <code>'</code>)</li> <li><code>\\</code> (literal <code>\</code>)</li> </ul>
Unicode	<code>'\u####'</code>	A Unicode code point. The <code>#</code> symbols are hexadecimal digits. A Unicode literal requires four digits.
null	<code>null</code>	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
<code>==</code>	<code>eq</code>	<code>1 == 1</code> <code>1 == 1.0</code> <code>1 eq 1</code>	Returns <code>true</code> if the operands are equal. This comparison is valid for all data types.
<code>!=</code>	<code>ne</code>	<code>1 != 2</code> <code>1 != true</code> <code>1 != '1'</code> <code>null != false</code> <code>1 ne 2</code>	Returns <code>true</code> if the operands are not equal. This comparison is valid for all data types.
<code>&lt;</code>	<code>lt</code>	<code>1 &lt; 2</code> <code>1 lt 2</code>	Returns <code>true</code> if the first operand is numerically less than the second. You must escape the <code>&lt;</code> operator to <code>&amp;lt;</code> ; to use it in Aura markup. Alternatively, you can use the <code>lt</code> operator.
<code>&gt;</code>	<code>gt</code>	<code>42 &gt; 2</code> <code>42 gt 2</code>	Returns <code>true</code> if the first operand is numerically greater than the second.
<code>&lt;=</code>	<code>le</code>	<code>2 &lt;= 42</code> <code>2 le 42</code>	Returns <code>true</code> if the first operand is numerically less than or equal to the second. You must escape the <code>&lt;=</code> operator to <code>&amp;lt;=</code> ; to use it in Aura markup. Alternatively, you can use the <code>le</code> operator.
<code>&gt;=</code>	<code>ge</code>	<code>42 &gt;= 42</code> <code>42 ge 42</code>	Returns <code>true</code> 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
<code>&amp;&amp;</code>	<code>isEnabled &amp;&amp; hasPermission</code>	Returns <code>true</code> if both operands are individually true. You must escape the <code>&amp;&amp;</code> operator to <code>&amp;amp; &amp;amp;</code> ; to use it in Aura markup. Alternatively, you can use the <code>and()</code> function and pass it two arguments. For example, <code>and(isEnabled, hasPermission)</code> .
<code>  </code>	<code>hasPermission    isRequired</code>	Returns <code>true</code> if either operand is individually true.
<code>!</code>	<code>!isRequired</code>	Unary operator. Returns <code>true</code> if the operand is false. This operator should

Operator	Usage	Description
		not be confused with the <code>!</code> delimiter used to start an expression in <code>{!}</code> . You can combine the expression delimiter with this negation operator to return the logical negation of a value, for example, <code>{!!true}</code> returns <code>false</code> .

## 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
<code>true</code>	<code>true</code>	A boolean <code>true</code> value.
<code>false</code>	<code>false</code>	A boolean <code>false</code> value.

## Conditional Operator

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

Operator	Usage	Description
<code>? :</code>	<code>(1 != 2) ? "Obviously" : "Black is White"</code>	The operand before the <code>?</code> 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 Aura 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
<code>add</code>	<code>concat</code>	<code>add(1, 2)</code>	Adds the first argument to the second.	<code>+</code>
<code>sub</code>	<code>subtract</code>	<code>sub(10, 2)</code>	Subtracts the second argument from the first.	<code>-</code>
<code>mult</code>	<code>multiply</code>	<code>mult(2, 10)</code>	Multiplies the first argument by the second.	<code>*</code>

Function	Alternative	Usage	Description	Corresponding Operator
<code>div</code>	<code>divide</code>	<code>div(4,2)</code>	Divides the first argument by the second.	<code>/</code>
<code>mod</code>	<code>modulus</code>	<code>mod(5,2)</code>	Returns the integer remainder resulting from dividing the first argument by the second.	<code>%</code>
<code>abs</code>		<code>abs(-5)</code>	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, <code>abs(-5)</code> is 5.	None
<code>neg</code>	<code>negate</code>	<code>neg(100)</code>	Reverses the sign of the argument. For example, <code>neg(100)</code> is -100.	<code>-</code> (unary)

## String Functions

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

## Array Functions

Function	Alternative	Usage	Description	Corresponding Operator
<code>length</code>		<code>myArray.length</code>	Returns the length of the array.	

## 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
<code>equals</code>	<code>equals(1,1)</code>	Returns <code>true</code> if the specified arguments are equal. The arguments can be any data type.	<code>==</code> or <code>eq</code>
<code>notequals</code>	<code>notequals(1,2)</code>	Returns <code>true</code> if the specified arguments are not equal. The arguments can be any data type.	<code>!=</code> or <code>ne</code>

Function	Usage	Description	Corresponding Operator
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
lessthanequal	lessthanequal(1,2)	Returns true if the first argument is numerically less than or equal to the second argument.	<= or le
greaterthanequal	greaterthanequal(2,1)	Returns true if the first argument is numerically greater 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.	&&
or	or(hasPermission, hasVIPPass)	Returns true if either one of the arguments is true.	
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)

# Chapter 5

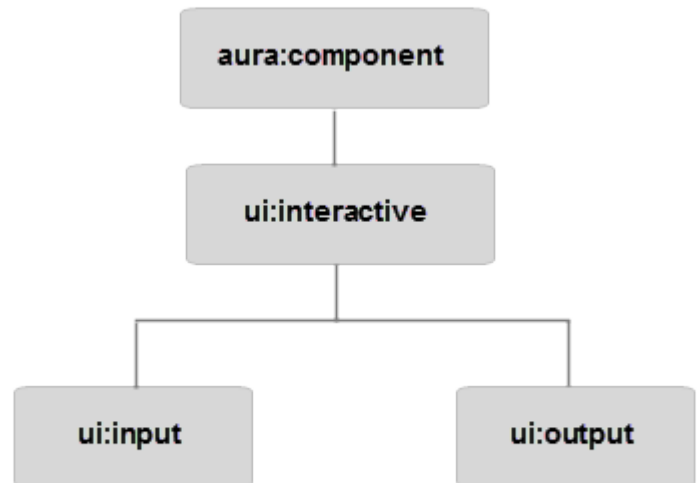
## User Interface Overview

---

### In this chapter ...

- [Input Components Overview](#)
- [Horizontal Layouts](#)
- [Vertical Layouts](#)
- [Working with Auto-Complete](#)
- [Creating Lists](#)

Aura provides common user interface components in the `ui` namespace. All of these components extend either `aura:component` or a child component of `aura:component`. `aura:component` is an abstract component that provides a default rendering implementation. Interactive user interface components such as `ui:input` and `ui:output` extend `ui:interactive`, which provides common user interface events like keyboard and mouse interactions. Each component can be styled and extended accordingly.



### See Also:

[Input Components Overview](#)  
[Components](#)  
[Component Bundles](#)

## Input Components Overview

Users interact with your app through input elements to select or make an input. Aura provides a range of input elements such as text fields, buttons, checkboxes, and so on.

`ui:input` provides child components, such as `ui:inputText` and `ui:inputCheckbox`, which correspond to common input elements. Each of these components support various input events, simplifying event handling for user interface events.

### Using the Input Components

To use Aura's input component in your own component, add them to your `.cmp` or `.app` file. This example is a basic set up of a text field and button.

```
<ui:inputText label="Name" aura:id="name" value="" placeholder="First, Last"/>
<ui:outputText aura:id="nameOutput" value=""/>
<ui:button aura:id="outputButton" label="Submit" press="{!c.getInput}"/>
```

The `ui:outputText` component acts as a placeholder for the output value of its corresponding `ui:inputText` component. The value in the `ui:outputText` component can be set with the following client-side controller action.

```
getInput : function(cmp, event) {
    var fullName = cmp.find("name").get("v.value");
    var outName = cmp.find("nameOutput");
    outName.set("v.value", fullName);
}
```

These are the input components that Aura provides out-of-the-box.

Input Type	Description	Related Components
Button	An actionable button that can be pressed or clicked.	<code>ui:button</code>
Checkbox	A selectable option that supports multiple selections.	<code>ui:inputCheckbox</code> <code>ui:outputCheckbox</code>
Field-level error	An error text that is displayed when a field-level error occurs.	<code>ui:inputDefaultError</code> <code>ui:message</code>
Select list	A list of options for single or multiple selection.	<code>ui:inputSelect</code> <code>ui:inputSelectOption</code> <code>ui:inputSelectOptionGroup</code> <code>ui:outputSelect</code>

These are the common field components you can use with Aura.

Field Type	Description	Related Components
Date and time	An input field for entering date and time.	<code>ui:inputDate</code> <code>ui:inputDateTime</code> <code>ui:outputDate</code>

Field Type	Description	Related Components
		ui:outputDateTime
Number	An input field for entering a numerical value.	ui:inputNumber ui:outputNumber
Text	An input field for entering single line of text.	ui:inputText ui:outputText
Text Area	An input field for entering multiple lines of text.	ui:inputTextArea ui:outputTextArea

## Buttons

A button is clickable and actionable, providing a textual label, an image, or both. You can create a button in three different ways:

- Text-only Button

```
<ui:button label="Find" />
```

- Image-only Button

```
<ui:button iconImgSrc="/auraFW/resources/aura/images/search.png" label="Find"
labelDisplay="false"/>
```

- Button with Text and Image

```
<ui:button label="Find" iconImgSrc="/auraFW/resources/aura/images/search.png"/>
```

## HTML Rendering

The markup for a button with text and image results in the following HTML.

```
<button class="default uiBlock uiButton" accesskey type="button">
  
  <span class="label bBody truncate" dir="ltr">Find</span>
</button>
```

## Working with Click Events

The `press` event on the `ui:button` component is fired when the user clicks the button. In the following example, `press="{!c.getInput}"` calls the client-side controller action with the function name, `getInput`, which outputs the input text value.

```
<aura:component>
  <ui:inputText aura:id="name" label="Enter Name:" placeholder="Your Name" />
  <ui:button aura:id="button" label="Click me" press="{!c.getInput}" />
</aura:component>
```



```
<ui:outputText aura:id="outName" value="" class="text"/>
</aura:component>
```

```
/* Client-side controller */
({
  getInput : function(cmp, evt) {
    var myName = cmp.find("name").get("v.value");
    var myText = cmp.find("outName");
    var greet = "Hi, " + myName;
    myText.set("v.value", greet);
  }
})
```

## Controlling Propagation

To control propagation of DOM events, use the `stopPropagation` attribute. This example toggles propagation on a `ui:button` component.

```
<aura:component>
  <aura:attribute name="propagation" type="Boolean" default="false"/>
  <div onclick="{!c.handleWrapperClick}">
    <ui:button press="{!c.handleClick}" stopPropagation="{!v.propagation}" label="Aura
    Button"/>
  </div><br/>
  Propagation status: {! v.propagation ? 'OFF' : 'ON'}<br/>
  <ui:button press="{!c.togglePropagation}" label="Toggle Propagation"/>
</aura:component>
```

```
/* Client-side controller */
({
  handleClick: function(cmp, event, helper) {
    console.log(event);
  },
  handleWrapperClick: function(cmp, event, helper) {
    alert('Click propagated to wrapper');
  },
  togglePropagation: function(cmp, event, helper) {
    cmp.set('v.propagation', !cmp.get('v.propagation'));
  }
})
```

## Styling Your Buttons

The `ui:button` component is customizable with regular CSS styling. In the CSS file of your component, add the following class selector.

```
.THIS.uiButton {
  margin-left: 20px;
}
```

Note that no space is added in the `.THIS.uiButton` selector if your button component is a top-level element.

To override the styling for all `ui:button` components in your app, in the CSS file of your app, add the following class selector.

```
.THIS .uiButton {
    margin-left: 20px;
}
```

**See Also:**  
[Handling Events with Client-Side Controllers](#)  
[CSS in Components](#)

**Date and Time Fields**

Date and time fields provide client-side localization, date picker support, and support for common keyboard and mouse events. If you want to render the output from these field components, use the respective `ui:output` components. For example, to render the output for the `ui:inputDate` component, use `ui:outputDate`.

Date and Time fields are represented by the following components.

Field Type	Description	Related Components
Date	An input field for entering a date of type text.	<code>ui:inputDate</code> <code>ui:outputDate</code>
Date and Time	An input field for entering a date and time of type text.	<code>ui:inputDateTime</code> <code>ui:outputDateTime</code>

**Using the Date and Time Fields**

This is a basic set up of a date field with a date picker.

```
<ui:inputDate aura:id="dateField" label="Birthday" value="2000-01-01"
displayDatePicker="true"/>
```

This example results in the following HTML.

```
<div class="uiInput uiInputDate">
  <label class="uiLabel-left uiLabel" for="globalId">
    <span>Birthday</span>
  </label>
  <input placeholder="MMM d, yyyy" type="text" id="globalId" class="uiInput uiInputDate">
  <a class="datePicker-openIcon" aria-haspopup="true">
    <span class="assistiveText">Date Picker</span>
  </a>
  <div class="uiDatePicker">
    <!--Date picker set to visible when icon is clicked-->
  </div>
</div>
```

## Localizing the Date and Time

The following code is a basic set up of a date and time field with client-side localization, which renders as `Mai 8, 2013 9:00:00 AM`.

```
<ui:outputDateTime langLocale="de" timezone="Europe/Berlin" value="2013-05-08"/>
```

## Styling Your Date and Time Fields

You can style the appearance of your date and time field and output in the CSS file of your component.

The following example provides styles to a `ui:inputDateTime` component with the `myStyle` selector.

```
<!-- Component markup -->
<ui:inputDateTime class="myStyle" label="Date" displayDatePicker="true"/>

/* CSS */
.THIS .myStyle {
  border: 1px solid #dce4ec;
  border-radius: 4px;
}
```

## See Also:

[Input Component Labels](#)

[Handling Events with Client-Side Controllers](#)

[Localization](#)

[CSS in Components](#)

## Number Fields

Number fields can contain a numerical value. They support client-side formatting, localization, and common keyboard and mouse events.

If you want to render the output from these field components, use the respective `ui:output` components. For example, to render the output for the `ui:inputNumber` component, use `ui:outputNumber`.

Number fields are represented by the following components.

Field Type	Description	Related Components
Number	An input field for entering a numerical value.	<code>ui:inputNumber</code> <code>ui:outputNumber</code>
Currency	An input field for entering a numerical currency value.	<code>ui:inputCurrency</code> <code>ui:outputCurrency</code>
Percentage	An input field for entering a numerical percentage value.	<code>ui:inputPercent</code> <code>ui:outputPercent</code>
Range	A slider for numerical input.	<code>ui:inputRange</code>

## Using the Number Fields

This example shows a basic set up of a percentage number field, which displays 50% in the field.

```
<ui:label label="Discount" for="discountField"/>
<ui:inputPercent aura:id="discountField" value="0.5"/>
```

This is a basic set up of a range input, with the min and max attributes.

```
<ui:label label="Quantity" for="qtyField"/>
<ui:inputRange aura:id="qtyField" min="1" max="10"/>
```

`ui:label` provides a text label for the corresponding field.

These examples result in the following HTML.

```
<label for="globalId" class="uiLabel"><span>Discount</span></label>
<input aria-describedby max="9999999999999999" step="1" placeholder type="text"
min="-9999999999999999" id="globalId" class="uiInput uiInputText uiInputNumber uiInputPercent">
```

```
<label for="globalId" class="uiLabel"><span>Quantity</span></label>
<input max="10" step="1" type="range" min="1" id="globalId" class="uiInput uiInputText
uiInputNumber uiInputRange">
```

## Returning a Valid Number

The value of the `ui:inputNumber` component expects a valid number and won't work with commas. If you want to include commas, use `type="Integer"` instead of `type="String"`.

This example returns 100,000.

```
<aura:attribute name="number" type="Integer" default="100,000"/>
<ui:inputNumber label="Number" value="{!v.number}"/>
```

This example also returns 100,000.

```
<aura:attribute name="number" type="String" default="100000"/>
<ui:inputNumber label="Number" value="{!v.number}"/>
```

## Formatting and Localizing the Number Fields

The `format` attribute determines the format of the number input. The Locale default format is used if none is provided. The following code is a basic set up of a number field, which displays 10,000.00 based on the provided `format` attribute.

```
<ui:label label="Cost" for="costField"/>
<ui:inputNumber aura:id="costField" format="#,##0,000.00#" value="10000"/>
```

The following code is a basic set up of a percentage field with client-side formatting, which displays 14.000% based on the provided `format` attribute.

```
<ui:label label="Growth" for="pField"/>
<ui:outputPercent aura:id="pField" value="0.14" format=".000%"/>
```

The following code is a basic set up of a currency field with localization, which displays £10.00 based on the provided `currencySymbol` and `format` attributes. You can also set the `currencyCode` attribute with an ISO 4217 currency code, such as USD or GBP.

```
<ui:outputCurrency value="10" currencySymbol="£" format="¤.00" />
```

Styling Your Number Fields

You can style the appearance of your number field and output. In the CSS file of your component, add the corresponding class selectors. The following class selectors provide styles to the string rendering of the numbers. For example, to style the `ui:inputCurrency` component, use `.THIS.uiInputCurrency`.

```
.THIS.uiInputNumber { //CSS declaration }
.THIS.uiInputCurrency { //CSS declaration }
.THIS.uiInputPercentage { //CSS declaration }
.THIS.uiInputRange { //CSS declaration }
```

The following example provides styles to a `ui:inputNumber` component with the `myStyle` selector.

```
<!-- Component markup -->
<ui:inputNumber class="myStyle" label="Amount" placeholder="0" />

/* CSS */
.THIS .myStyle {
  border: 1px solid #dce4ec;
  border-radius: 4px;
}
```

See Also:

- [Input Component Labels](#)
- [Handling Events with Client-Side Controllers](#)
- [Localization](#)
- [CSS in Components](#)

Text Fields

A text field can contain alphanumerical characters and special characters. They inherit the functionalities and events from `ui:inputText` and `ui:input`, including `placeholder` and `size` and common keyboard and mouse events. If you want to render the output from these field components, use the respective `ui:output` components. For example, to render the output for the `ui:inputPhone` component, use `ui:outputPhone`.

Text fields are represented by the following components.

Field Type	Description	Related Components
Email	An input field for entering an email address.	ui:inputEmail ui:outputEmail
Phone	An input field for entering a phone number.	ui:inputPhone ui:outputPhone

Field Type	Description	Related Components
Rich Text	An input field for entering rich text	ui:inputRichText ui:outputRichText
Search	An input field for entering a search term.	ui:inputSearch
Text	An input field for entering a single-line text.	ui:inputText ui:outputText
Text Area	An input field for entering multiple-line text.	ui:inputTextArea ui:outputTextArea
URL	An input field for entering a URL.	ui:inputURL ui:outputURL

### Using the Text Fields

This is a basic set up of an email field.

```
<ui:inputEmail aura:id="email" label="Email" placeholder="abc@email.com"/>
```

This example results in the following HTML.

```
<div class="uiInput uiInputText uiInputEmail">
  <label class="uiLabel-left uiLabel" for="globalId">
    <span>Email</span>
  </label>
  <input placeholder="abc@email.com" type="email" id="globalId" class="uiInput uiInputText uiInputEmail">
</div>
```

### Providing Auto-complete Suggestions in Text Fields

Auto-complete is available with the `ui:autocomplete` component, which uses a text or text area of its own. To use a text area, set the `inputType="inputTextArea"`. The default is `inputText`.

### Styling Your Text Fields

You can style the appearance of your text field and output. In the CSS file of your component, add the corresponding class selectors.

The following class selectors provide styles to the string rendering of the text. For example, to style the `ui:inputPhone` component, use `.THIS .uiInputPhone`.

```
.THIS.uiinputEmail { //CSS declaration }
.THIS.uiinputphone { //CSS declaration }
.THIS.uiinputtext { //CSS declaration }
.THIS.uiinputtextarea { //CSS declaration }
.THIS.uiinputurl { //CSS declaration }
```

The following example provides styles to a `ui:inputText` component with the `myStyle` selector.

```
<!-- Component markup-->
<ui:inputText class="myStyle" label="Name"/>
```

```
/* CSS */
.THIS .myStyle {
  border: 1px solid #dce4ec;
  border-radius: 4px;
}
```

## See Also:

[Rich Text Fields](#)

[Input Component Labels](#)

[Handling Events with Client-Side Controllers](#)

[Localization](#)

[CSS in Components](#)

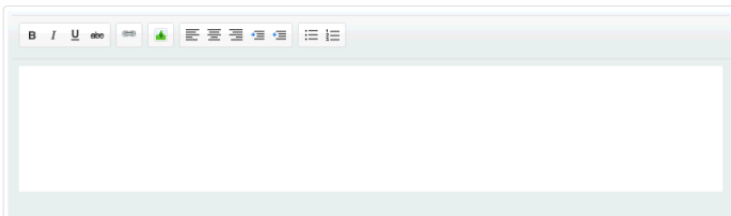
## Rich Text Fields

`ui:inputRichText` is an input field for entering rich text. The following code shows a basic implementation of this component, which is rendered as a text area and button. A button click runs the client-side controller action that returns the input value in a `ui:outputRichText` component. In this case, the value returns “Aura” in bold, and “input rich text demo” in red.

```
<!--Rich text demo-->
<ui:inputRichText isRichText="false" aura:id="inputRT" label="Rich Text Demo"
labelPosition="hidden"
  cols="50" rows="5" value="&lt;b&gt;Aura&lt;/b&gt;; &lt;span style='color:red'&gt;input
rich text demo&lt;/span&gt;"/>
<ui:button aura:id="outputButton"
  buttonText="Click to see what you put into the rich text field"
  label="Display" press="{!c.getInput}"/>
<ui:outputRichText aura:id="outputRT" value=" "/>
```

```
/*Client-side controller*/
getInput : function(cmp) {
  var userInput = cmp.find("inputRT").get("v.value");
  var output = cmp.find("outputRT");
  output.set("v.value", userInput);
}
```

In this demo, the `isRichText="false"` attribute replaces the component with the `ui:inputTextArea` component. The WYSIWYG rich text editor is provided when this attribute is not set, as shown below.



The width and height of the rich text editor are independent of those on the `ui:inputTextArea` component. To set the width and height of the component when you set `isRichText="false"`, use the `cols` and `rows` attributes. Otherwise, use the `width` and `height` attributes.

## See Also:

[Text Fields](#)

## Checkboxes

Checkboxes are clickable and actionable, and they can be presented in a group for multiple selection. You can create a checkbox with `ui:inputCheckbox`, which inherits the behavior and events from `ui:input`. The `value` and `disabled` attributes control the state of a checkbox, and events such as `click` and `change` determine its behavior. Events must be used separately on each checkbox.

Here are several basic ways to set up a checkbox.

### Checked

To select the checkbox, set `value="true"`. Alternatively, `value` can take in a value from a model.

```
<ui:inputCheckbox value="true"/>

<!--Initializing the component-->
<ui:inputCheckbox aura:id="inCheckbox" value="{!m.checked}"/>

//Initializing with a model
public Boolean getChecked() {
    return true;
}
```

The model is in a Java class specified by the `model` attribute on the `aura:component` tag.

### Disabled State

```
<ui:inputCheckbox disabled="true" label="Select" labelPosition="left" />
```

The previous example results in the following HTML.

```
<label class="uiLabel-left uiLabel" for="globalId"><span>Select</span></label>
<input disabled="disabled" type="checkbox" id="globalId" class="uiInput uiInputCheckbox">
```

### Working with Events

Common events for `ui:inputCheckbox` include the `click` and `change` events. For example, `click="{!c.done}"` calls the client-side controller action with the function name, `done`.

The following code crosses out the checkbox item.

```
<!--The checkbox-->
<ui:inputCheckbox label="Cross this out" click="{!c.crossout}" class="line"
labelPosition="right"/>

<!--The controller action-->
crossout : function(cmp, event){
```



```
var elem = event.getSource().getElement();
$A.util.toggleClass(elem, "done");
}
```

## Styling Your Checkboxes

The `ui:inputCheckbox` component is customizable with regular CSS styling. This example shows a checkbox with a custom



```
<ui:inputCheckbox labelClass="check" label="Select?" value="true" labelPosition="right" />
```

The following CSS style replaces the default checkbox with the given image.

```
.THIS input[type="checkbox"] {
    display: none;
}

.THIS .check span {
    margin: 20px;
}

.THIS input[type="checkbox"]+label {
    display: inline-block;
    width: 20px;
    height: 20px;
    vertical-align: middle;
    background: url('images/checkbox.png') top left;
    cursor: pointer;
}

.THIS input[type="checkbox"]:checked+label {
    background: url('images/checkbox.png') bottom left;
}
```

## See Also:

[Java Models](#)

[Handling Events with Client-Side Controllers](#)

[CSS in Components](#)

## Field-level Errors

Field-level errors are displayed when a validation error occurs on the field after a user input. Aura creates a default error component, `ui:inputDefaultError`, which provides basic events such as `click` and `mouseover`. See [Validating Fields](#) for more information.

Alternatively, you can use `ui:message` for field-level errors.

### Invalid password

Your password should be at least 6 alphanumeric characters long.

Here are a few basic ways to set up a field-level error using `ui:message`. The error message is persistent by default but you can set `closable="true"` if you want the user to be able to close it.

### Visible

```
<ui:message title="Invalid password" severity="error" closable="true">
  Your password should be at least 6 alphanumeric characters long.
</ui:message>
```

### Visible on error condition

```
<!--The ui:message component-->
<aura:renderIf isTrue="{!v.invalidPW}">
  <ui:message title="Invalid password" severity="error" closable="true">
    Your password should be at least 6 alphanumeric characters long.
  </ui:message>
</aura:renderIf>
```

```
<!--The client-side controller action-->
var pw = cmp.find("inPW").get("v.value");
if (pw.length >= 6) {
  outPW.set("v.value", pw);
  cmp.set("v.invalidPW", false);
} else {
  cmp.set("v.invalidPW", true);
}
```

`aura:renderIf` conditionally renders the body if the `isTrue` attribute evaluates to true.

### Working with Events

Common events for `ui:message` include the `click` and `mouseover` events. For example, `click="{!c.revalidate}"` calls the client-side controller action with the function name, `revalidate`, when a user clicks on the error message.

### Styling Your Field-Level Errors

The `ui:message` component is customizable with regular CSS styling. The following CSS sample replaces the default border. This component can be used with varying severity levels, which uses different styles. To compare the severity levels and styles, see the [ui:message demo](#).

Alternatively, use the `class` attribute to specify your own CSS class.

### See Also:

[Handling Events with Client-Side Controllers](#)

[CSS in Components](#)

## Drop-down Lists

Drop-down lists display a dropdown menu with available options. Both single and multiple selections are supported. You can create a drop-down list using `ui:inputSelect`, which inherits the behavior and events from `ui:input`.

Here are a few basic ways to set up a drop-down list.

For multiple selections, the default number of options displayed can be specified by the `size` attribute.

## Single Selection

```
<ui:inputSelect>
  <ui:inputSelectOptionGroup label="Group 1">
    <ui:inputSelectOption text="Red"/>
    <ui:inputSelectOption text="Green" value="true"/>
    <ui:inputSelectOption text="Blue"/>
  </ui:inputSelectOptionGroup>
  <ui:inputSelectOptionGroup label="Group 2">
    <ui:inputSelectOption text="Cyan"/>
    <ui:inputSelectOption text="Magenta"/>
    <ui:inputSelectOption text="Yellow"/>
  </ui:inputSelectOptionGroup>
</ui:inputSelect>
```

This example results in the following HTML.

```
<select size="1" id="globalId" class="uiInput uiInputSelect">
  <optgroup label="Group 1" class="uiInputSelectOptionGroup">
    <option value="Red" class="uiInputSelectOption">Red</option>
    <!--more option tags here-->
  </optgroup>
  <!--more optgroup tags here-->
</select>
```

## Multiple Selection

```
<ui:inputSelect multiple="true" size="5">
  <ui:inputSelectOptionGroup label="Group 1">
    <ui:inputSelectOption text="Red"/>
    <ui:inputSelectOption text="Green" value="true"/>
    <ui:inputSelectOption text="Blue"/>
  </ui:inputSelectOptionGroup>
  <ui:inputSelectOptionGroup label="Group 2">
    <ui:inputSelectOption text="Cyan"/>
    <ui:inputSelectOption text="Magenta"/>
    <ui:inputSelectOption text="Yellow"/>
  </ui:inputSelectOptionGroup>
</ui:inputSelect>
```

The default selected value is specified by `value="true"`. Each option is represented by `ui:inputSelectOption`, which is nested in a `ui:inputSelectOptionGroup` component.

## Generating Options Dynamically

To generate the options dynamically, use the method shown below.

```
<aura:component>
  <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
  <ui:inputSelect label="Select me:" class="dynamic" aura:id="InputSelectDynamic"/>
</aura:component>
```

The following client-side controller generates options using the `options` attribute on the `ui:inputSelect` component. `v.options` takes in the list of objects and converts them into list options. Although the sample code generates the options

during initialization, the list of options can be modified anytime when you manipulate the list in `v.options`. The component automatically updates itself and rerenders with the new options.

```
((
  doInit : function(cmp) {
    var opts = [
      { class: "optionClass", label: "Option1", value: "opt1", selected: "true" },
      { class: "optionClass", label: "Option2", value: "opt2" },
      { class: "optionClass", label: "Option3", value: "opt3" }
    ];
    cmp.find("InputSelectDynamic").set("v.options", opts);
  }
})
```



**Note:** `aura:iteration` is not supported for `ui:inputSelect`. We recommend using a client-side controller or model to generate your options iteratively.

In the preceding demo, the `opts` object constructs `InputOption` objects to create the `ui:inputSelectOptions` components within `ui:inputSelect`.

### Generating Options with a Model

Display a list of options from a model by using the format `<ui:inputSelect options="{!m.selectOptions}"/>`, and creating the list of options in the model using `ArrayList<InputOption>`.

The following code shows a model that generates a list of options for a `ui:inputSelect` component.

```
@Model
public class SizeModel {

    @AuraEnabled
    public List<InputOption> getSizes() {
        ArrayList<InputOption> a = new ArrayList<InputOption>(3);
        InputOption m1 = new InputOption("Small", "s", false, "Small");
        a.add(m1);
        InputOption m2 = new InputOption("Medium", "m", false, "Medium");
        a.add(m2);
        InputOption m3 = new InputOption("Large", "l", false, "Large");
        a.add(m3);
        return a;
    }
}
```

The following component code displays the list of options using the given model.

```
<aura:component model="java://org.auraframework.docs.SizeModel">
  <aura:attribute name="sizes" type="List" description="A list input options."/>
  <ui:inputSelect label="Size" options="{!m.sizes}"/>
</aura:component>
```

The `InputOption` object has these parameters.

Parameter	Type	Description
label	String	The label of the option to display on the user interface.
name	String	The name of the option.
selected	boolean	Indicates whether the option is selected.

Parameter	Type	Description
value	String	The value of this option.

### Working with Events

Common events for `ui:inputSelect` include the `change` and `click` events. For example, `change="{!c.onSelectChange}"` calls the client-side controller action with the function name, `onSelectChange`, when a user changes a selection.

### Styling Your Field-level Errors

The `ui:inputSelect` component is customizable with regular CSS styling. The following CSS sample adds a fixed width to the drop-down menu.

```
.THIS.uiInputSelect {
    width: 200px;
}
```

Alternatively, use the `class` attribute to specify your own CSS class.

### See Also:

[Handling Events with Client-Side Controllers](#)

[CSS in Components](#)

## Horizontal Layouts

`ui:block` provides a horizontal layout for your components. It extends `aura:component` and is an actionable component. It is useful for laying out your labels, fields, and buttons or any groups of components in a row.

Here is a basic set up of a horizontal layout. The following sample code creates a horizontal view of an image, text field, and a button. The `ui:inputText` component renders in between the `left` and `right` attributes.

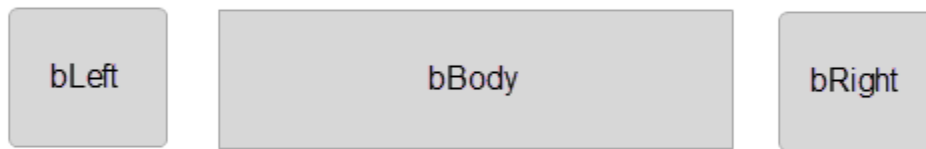
```
<ui:block>
    <aura:set attribute="left">
        <ui:image src="/auraFW/resources/aura/images/search.png" alt="bLeft" />
    </aura:set>
    <aura:set attribute="right">
        <ui:button label="Submit"/>
    </aura:set>
    <ui:inputText label="Text" labelPosition="hidden" />
</ui:block>
```

### Working with Events

Common events for `ui:block` include the `click` and `mouseover` events. For example, `click="{!c.enable}"` calls the client-side controller action with the function name, `enable`, when a user clicks anywhere in the layout.

### Styling Your Horizontal Layouts

`ui:block` is customizable with regular CSS styling. The output is rendered in `div` tags with the `bLeft`, `bRight`, and `bBody` classes.



The following CSS class styles the `bLeft` class on the `ui:block`.

```
.THIS.uiBlock .bLeft { //CSS declaration }
```

Alternatively, use the `class` attribute to specify your own CSS class.

### See Also:

[Handling Events with Client-Side Controllers](#)

[CSS in Components](#)

## Vertical Layouts

`ui:vbox` provides a vertical layout for your components. It extends `aura:component` and is an actionable component. It is useful for laying out groups of components vertically on a page.

Here is a basic set up of a vertical layout. The following sample code creates a vertical view of a header, body, and footer. The body of the component renders in between the `north` and `south` attributes.

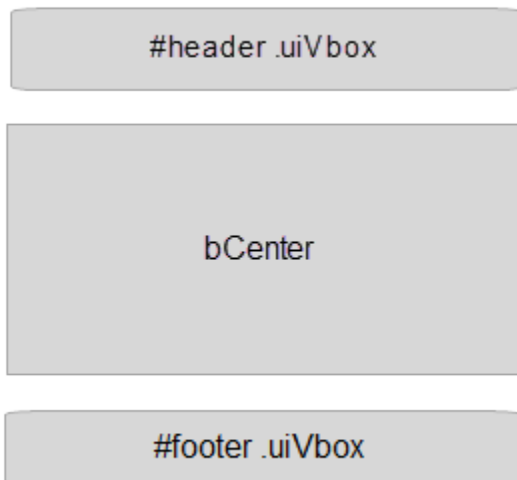
```
<ui:vbox>
  <aura:set attribute="north">
    <div id="header">Header</div>
  </aura:set>
  <aura:set attribute="south">
    <div id="footer">Footer</div>
  </aura:set>
  body
</ui:vbox>
```

### Working with Events

Common events for `ui:vbox` include the `click` and `mouseover` events. For example, `click="{!c.enable}"` calls the client-side controller action with the function name, `enable`, when a user clicks anywhere in the layout.

### Styling Your Vertical Layouts

`ui:vbox` is customizable with regular CSS styling. Given the above example, the output is rendered in `<div id="header" class="uiVbox">` and `<div id="footer" class="uiVbox">` tags, with the footer rendered in the bottom.



The following CSS class styles the header element in the north attribute.

```
.THIS #header { //CSS declaration }
```

### See Also:

[Handling Events with Client-Side Controllers](#)

[CSS in Components](#)

## Working with Auto-Complete

`ui:autocomplete` displays suggestions as users type in a text field. Data for this component is provided by a server-side model. This component provides its own text field and text area component. The default is a text field but you can change it to a text area by setting `inputType="inputTextArea"`.

Here is a basic set up of the auto-complete component with a default input text field.

```
<ui:autocomplete aura:id="autoComplete" optionVar="row"
  matchDone="{!c.handleMatchDone}"
  inputChange="{!c.handleInputChange}"
  selectListOption="{!c.handleSelectOption}">
  <aura:set attribute="dataProvider">
    <demo:dataProvider/>
  </aura:set>
  <aura:set attribute="listOption">
    <ui:autocompleteOption label="{!row.label}" keyword="{!row.keyword}"
      value="{!row.value}" visible="{!row.visible}"/>
  </aura:set>
</ui:autocomplete>
```

### Working with Events

Common events for `ui:autocomplete` include the `fetchData`, `inputChange`, `matchDone`, and `selectListOption` events. The behaviors for these events can be configured as desired.

**fetchData**

Fire the `fetchData` event if you want to fetch data through the data provider. For example, you can fire this event in the `inputChange` event when the input value changes. The `ui:autocomplete` component automatically matches text on the new data.

**inputChange**

Use the `inputChange` event to handle an input value change. Get the new value with `event.getParam("value")`. The following code handles a text match on existing data.

```
var matchEvt = acCmp.get("e.matchText");
matchEvt.setParams({
    keyword: event.getParam("value")
});
matchEvt.fire();
```

**matchDone**

Use the `matchDone` event to handle when a text matching has completed, regardless if a match has occurred. You can retrieve the number of matches with `event.getParam("size")`.

**selectListOption**

Use the `selectListOption` event to handle when a list option is selected. Get the options with `event.getParam("option")`; . This event is fired by the `ui:autocompleteList` component when a list option is selected.

**Providing Data to the Auto-complete Component**

In the basic set up above, `demo:dataProvider` provides the list of data to be displayed as suggestions when a text match occurs. `demo:dataProvider` extends `ui:dataProvider` and takes in a server-side model.

The following code is a sample data provider for the `ui:autocomplete` component.

```
<aura:component extends="ui:dataProvider"
    model="java://org.auraframework.impl.java.model.TestJavaModel">
    <aura:attribute name="dataType" type="String"/>
</aura:component>
```

In the client-side controller or helper function of your data provider, fire the `onchange` event on the parent `ui:dataProvider` component. This event handles any data changes on the list.

```
var data = component.get("m.listOfData");
var dataProvider = component.getConcreteComponent();
//Fire the onchange event in the ui:dataProvider component
this.fireDataChangeEvent(dataProvider, data);
```

See the data provider at `aura/src/test/components/uitest/testAutocompleteDataProvider` in the GitHub repo.

To learn how the data provider is retrieving data from the model, see the server-side model at `/aura-impl/src/test/java/org/auraframework/impl/java/model/TestJavaModel.java` in the [GitHub repo](#).



## Styling Your Auto-complete Component

The `ui:autocomplete` component is customizable with regular CSS styling. For example, if you're using the default text field component provided by `ui:autocomplete`, you can use the following CSS selector.

```
.THIS.uiInputText {
    //CSS declaration
}
```

If you're using the default text area component provided by `ui:autocomplete`, change the CSS selector to `.THIS.uiInputTextArea`. Alternatively, use the `class` attribute to specify your own CSS class.

### See Also:

[Handling Events with Client-Side Controllers](#)

[CSS in Components](#)

[Client-Side Runtime Binding of Components](#)

## Creating Lists

You can create lists in three different ways, using `aura:iteration`, `ui:list`, or `ui:infiniteList`. `aura:iteration` is used for simple lists and can take in data from a model.

`ui:list` and `ui:infiniteList` provide a paging interface to navigate lists. `ui:list` can be used for more robust list implementations that retrieves and display more data as necessary, with a data provider and a template for each list item. Additionally, use `ui:infiniteList` if you want a robust list implementation similar to `ui:list`, but with a handler that enables you to retrieve and display more data when the user reaches the bottom of the list.

Here is a basic set up of the `ui:list` component with a required data provider and template.

```
<ui:list itemVar="item">
    <aura:set attribute="dataProvider">
        <auradev:testDataProvider />
    </aura:set>
    <aura:set attribute="header">
        Item List
    </aura:set>
    <aura:set attribute="itemTemplate">
        <auradocs:demoListTemplate label="{!item.label}" />
    </aura:set>
</ui:list>
```

`itemVar` is a required attribute that is used to iterate over the items provided by the item template. In the above example, `{!item.label}` iterates over the items provided by the data provider and displays the labels.

The sample template, `auradocs:demoListTemplate` is as follows. This template is a row of text generated by the data provider.

```
<aura:component>
    <aura:attribute name="label" type="String"/>
    <div class="row">
        {!v.label}
    </div>
</aura:component>
```

## Working with List Events

`ui:list` and `ui:infiniteList` inherits from `ui:abstractList`. Common events for `ui:list` include user interface events like `click` events, and list-specific events like `refresh` and `triggerDataProvider`.

### refresh

The `refresh` event handles a list data refresh and fires the `triggerDataProvider` event. You can fire the `refresh` event by using the following sample code in your client-side controller action.

```
var listData = cmp.find("listData");
listData.get("e.refresh").fire();
```

### showMore

The `showMore` event in `ui:infiniteList` handles the fetching of your data and displays it. This event fires the `triggerDataProvider` event as well.

### triggerDataProvider

The `triggerDataProvider` event triggers the providing of data from a data provider. It is also run during component initialization and `refresh`. For example, you can use this event if you want to retrieve more data in a `ui:infiniteList` component.

```
cmp.set("v.currentPage", targetPage);
var listData = component.find("listData");
listData.get("e.triggerDataProvider").fire();
```

## Providing Data to the List Component

In the basic set up above, `auradocs:demoDataProvider` provides the list of data to the `ui:list` component. `auradocs:demoDataProvider` extends `ui:dataProvider` and takes in a server-side model.

The following code is the sample data provider, `auradocs:demoDataProvider`.

```
<aura:component extends="ui:dataProvider"
    model="java://org.auraframework.component.auradev.TestDataProviderModel"
    controller="java://org.auraframework.component.auradev.TestDataProviderController"
    description="A data provider for ui:list">
    <aura:handler name="provide" action="{!c.provide}"/>
</aura:component>
```

The `provide` event is fired on initialization by the parent `ui:abstractList` component. You can customize the `provide` event in your client-side controller. For example, the following code shows a sample `provide` helper function for a data provider.

```
var dataProvider = component.getConcreteComponent();
var action = dataProvider.get("c.getItems");

//Set the parameters for this action
action.setParams({
    "currentPage": dataProvider.get("v.currentPage"),
    "pageSize": dataProvider.get("v.pageSize")
    //Other ui:list or ui:infiniteList parameters
});

//Set the action callback
action.setCallback(this, function(action) {
    if (action.getState() === "SUCCESS") {
        var result = action.getReturnValue();
```

```
        this.fireDataChangeEvent(dataProvider, result);
    }
});
$A.enqueueAction(action);
```



**Note:** See the data provider at `aura-components/src/main/components/auradocs/demoDataProvider/` in the [GitHub repo](#).

To learn how the data provider is retrieving data from the model, see the server-side model at `aura-impl/src/main/java/org/auraframework/component/auradev/TestDataProviderModel.java`.

## Styling Your List Component

The `ui:list` component is customizable with regular CSS styling. For example, the sample template code above has `<div class="row">`. To apply CSS, you can use the following CSS selector in the template component.

```
.THIS .row{
    //CSS declaration
}
```

### See Also:

[Handling Events with Client-Side Controllers](#)

[CSS in Components](#)

# Chapter 6

## Using Labels

---

### In this chapter ...

- [\\$Label](#)
- [Input Component Labels](#)
- [Dynamically Populating Label Parameters](#)
- [Dynamically Creating Labels](#)
- [Customizing your Label Implementation](#)
- [Setting Label Values via a Parent Attribute](#)

Aura supports labels to enable you to separate field labels from your code.

## \$Label

Separating labels from source code makes it easier to translate and localize your applications. Use the `$Label` global value provider to access labels stored outside your code.

`$Label` doesn't have a default implementation but the `LocalizationAdapter` interface assumes that a label has a two-part name: a section name and a label name. This enables you to organize labels into sections with similar labels grouped together.

To customize the behavior of the `$Label` global value provider, see [Customizing your Label Implementation](#) on page 65.

Access a label using the dot notation, `$Label.<section>.<labelName>`; for example, `{!$Label.SocialApp.YouLike}`.

Each name must start with a letter or underscore so that the label can be accessed in an expression. For example, `{!$Label.1SocialApp.2YouLike}` is not valid because the section and label name each start with a number.

## 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.

```
<ui:inputNumber label="Pick a Number:" labelPosition="top" value="54" />
```

The label position can be hidden, top, right, or bottom. The default position is left.

### Using \$Label

Use the `$Label` global value provider to access labels stored in an external source. For example:

```
<ui:inputNumber label="{!$Label.Number.PickOne}" />
```

### Sourcing Labels from a Model

This example sources the labels from a model. The model has an `iterationItems` field, which is a collection of labels. Each item in the collection has a `label` and `value` attribute.

```
<aura:component model="java://org.auraframework.docs.LabelTestModel">
    <aura:iteration items="{!m.iterationItems}" var="item">
        <ui:inputText label="{!item.label}" value="{!item.value}" aura:id="iterate"/>
    </aura:iteration>
</aura:component>
```

### Separating Labels from Input Components

For design reasons, you might want a significant visual separation of an HTML `<label>` tag from its corresponding form element. In such a scenario, use the `ui:label` component to bind the label to the input component using the local ID, `aura:id`, of the input component.

This code sample shows how to bind a label using the `aura:id` of an input component.

```
<ui:label labelDisplay="false" for="myInput" label="My Input Text" />
<!-- HTML markup separating the label from the input component-->
<ui:inputText aura:id="myInput" value="Put your input here." />
```

To associate the `ui:label` tag with the input component, the `for` attribute in `ui:label` is set to the same value as the `aura:id` in the input component.

Note that setting `labelDisplay="false"` in `ui:label` hides the label from view but still exposes it to screen readers. For more information, refer to the `ui:label` component reference documentation.

### See Also:

[Dynamically Populating Label Parameters](#)

[Dynamically Creating Labels](#)

[Supporting Accessibility](#)

[Java Models](#)

## Dynamically Populating Label Parameters

The `aura:label` component accepts parameters, enabling you to dynamically populate placeholder values in labels.

The label component value attribute accepts one or more numbered parameters. This example substitutes the `{0}` parameter with an expression.

```
<aura:label value="{0} Members">
    {!v.numberOfMembers}
</aura:label>
```

This example shows the output and source of a label with a hard-coded expression value.

```
<aura:component>
    <aura:label value="Your balance is {0} points.">
        {!500}
    </aura:label>
</aura:component>
```

### Using `$Label`

You can dynamically populate parameters in a label using the global value provider, `$Label`. For example, if you have a `MySection.MyLabel` label set to `{0} Members`, you can provide a value for the parameter when you reference the label in `aura:label`.

```
<aura:label value="{!$Label.MySection.MyLabel}">
    {!v.numberOfMembers}
</aura:label>
```

If the `v.numberOfMembers` expression evaluates to 5, the output will be:

```
5 Members
```

You can 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.

This example shows the `MySection.MyLabel` label defined as "`{0} Members, {1} New Members, and {2} Guests`" in the label file.

```
<aura:label value="{!$Label.MySection.MyLabel}">
    {!v.numberOfMembers}
```

```

    {!v.numberofNewMembers}
    {!v.numberofGuests}
  </aura:label>

```

Assuming that `{!v.numberofMembers}` evaluates to 5, `{!v.numberofNewMembers}` evaluates to 2, and `{!v.numberofGuests}` evaluates to 8, the output is:

```
5 Members, 2 New Members, and 8 Guests
```

You can specify an Aura or HTML component as a parameter substitution value in the body of `aura:label`. This example shows how to include a link in a label by substituting the `{0}` parameter with the embedded `ui:outputURL` Aura component. The `$Label.MySection.LinkLabel` label is defined as `Label` with `link: {0}`.

```

<aura:label value="{!$Label.MySection.LinkLabel}">
  <ui:outputURL value="http://www.salesforce.com" label="Test Link"/>
</aura:label>

```

This example is similar to the previous one except that the label value is hard-coded and doesn't use the label provider.

```

<aura:component>
  <aura:label value="Label with link: {0}">
    <ui:outputURL value="http://www.salesforce.com" label="Test Link"/>
  </aura:label>
</aura:component>

```

This is equivalent to embedding the HTML anchor tag:

```

<aura:label value="{!$Label.MySection.LinkLabel}">
  <a href="http://www.salesforce.com">Test Link</a>
</aura:label>

```

## Embedding `aura:label` in Another Component

You can use an `aura:label` component with parameter substitutions as the label of another component. For example, you can use an `aura:label` component as the label of a `ui:button` component. Set the `labelDisplay` attribute to `false` so that the label attribute won't be rendered. The embedded label in `aura:label` is displayed instead.

This example embeds the label component from the previous example inside a `ui:button` component. The button label is taken from this embedded label component, which in turn contains an `ui:outputURL` component in its body for substituting a parameter with a link. `$Label.MySection.LinkLabel` is defined as `Label` with `link: {0}`.

```

<ui:button labelDisplay="false" label="Label for assistive text">
  <aura:label value="{!$Label.MySection.LinkLabel}">
    <ui:outputURL value="http://www.salesforce.com" label="Test Link"/>
  </aura:label>
</ui:button>

```

This example uses a hard-coded label value rather than a value from the label provider.

```

<aura:component>
  <ui:button labelDisplay="false" label="Label for assistive text">
    <aura:label value="Label with link: {0}">
      <ui:outputURL value="http://www.salesforce.com" label="Test Link"/>
    </aura:label>
  </ui:button>
</aura:component>

```

## Dynamically Creating Labels

You can dynamically create labels in JavaScript code. This can be useful when you need to use a label that is not known until runtime when it's dynamically generated.

This example retrieves a label from the server and binds the label value to a component.

```
$A.getGlobalValueProviders().getValue("$Label.Related_Lists.task_mode_today", cmp);
```

If the label value isn't already known on the client, then the label is fetched asynchronously from the server. If you specify the component parameter, the component will be updated with the label when it's retrieved from the server.

`$A.getGlobalValueProviders().getValue()` takes three parameters:

- A required expression
- An optional component
- An optional callback

This example dynamically constructs the label value by calling `$A.getGlobalValueProviders().getValue()` to update an attribute in a component.

### Component source

```
<aura:component render="client">
  <aura:attribute name="simplevalue1" type="String"/>
  <div>Dynamic label update: {!v.simplevalue1}</div>
  <ui:button press="{!c.getLabel}" label="Get Label" />
</aura:component>
```

### Client-side controller source

```
{
  getLabel: function(cmp, event) {
    var gvp = $A.getGlobalValueProviders();
    // Demonstrating dynamic construction of a label string.
    // This example is contrived but partialLabel could be
    // dynamically constructed in your code.
    var partialLabel = "task_mode_today";
    var dynLabel = gvp.getValue(
      "$Label" + ".Related_Lists." + partialLabel, cmp
    );
    cmp.set("v.simplevalue1", dynLabel);
  }
}
```

`<aura:attribute name="simplevalue1" type="String"/>` is a placeholder for the label. The attribute is dynamically updated with the label value, which triggers rerendering of the component.

Note that it is important in this example that the `$Label` value is dynamically concatenated in the JavaScript code. If you used a static label, such as `$Label.Related_Lists.task_mode_today`, instead, Aura would have simply pre-fetched the value for the static label and sent it to the client.

If you call `$A.getGlobalValueProviders().get()` instead of `$A.getGlobalValueProviders().getValue()`, the label will be fetched asynchronously from the server but the component won't be updated with the label value until you press the button again. This is because the return value is a `String` instead of a `SimpleValue`.



If the label value is already known on the client, you can use `$A.get("$Label.Related_Lists.task_mode_today")` as an alternative to `$A.getGlobalValueProviders().get("$Label.Related_Lists.task_mode_today")`. Both calls behave the same once the client knows the label value.

## Rendering Dynamic Labels

If the label is already known on the client, `$A.getGlobalValueProviders().get()` displays the label. If the value is not known, PROD mode displays an empty placeholder and all other modes return a placeholder containing the label expression. The placeholder is replaced with the label value when it's retrieved from the server.

## Testing Dynamic Labels

To inspect the label value after it is retrieved, use a callback function.

```
$A.getGlobalValueProviders().get("$Label" + ".Related_Lists" + ".task_mode_today", cmp,
    function(res) {
        $A.test.assertEquals("Today", res, "Failed: Wrong label value in callback");
    }
);
```

## Avoiding a Server Roundtrip

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 `$Label.Related_Lists.task_mode_today` and `$Label.Related_Lists.task_mode_tomorrow` label keys, you can add references to the labels in a JavaScript resource, such as a client-side controller or helper.

```
var preloadHint1 = $A.get("$Label.Related_Lists.task_mode_today");
var preloadHint2 = $A.get("$Label.Related_Lists.task_mode_tomorrow");
```

## See Also:

[Using JavaScript](#)

[Input Component Labels](#)

[Dynamically Populating Label Parameters](#)

[Customizing your Label Implementation](#)

[Modes Reference](#)

# Customizing your Label Implementation

You can customize where your app reads labels from by overriding the default label adapter. Your label adapter implementation encapsulates the details of finding and returning labels defined outside the application code. Typically, labels are defined separately from the source code to make localization of labels easier.

To provide a label adapter implementation, implement the `LocalizationAdapter` interface with the following two methods.

```
public class MyLocalizationAdapterImpl implements LocalizationAdapter {

    @Override
    public String getLabel(String section, String name, Object... params) {
        // Return specified label.
    }

    @Override
```

```

    public boolean labelExists(String section, String name) {
        // Return true if the label exists; otherwise false.
    }
}

```

The `getLabel` method contains the implementation for finding the specified label and returning it. Here is a description of its parameters:

Parameter	Description
String <i>section</i>	The section in the label definition file where the label is defined. This assumes your label name has two parts (section.name). This parameter can be <code>null</code> depending on your label system implementation.
String <i>name</i>	The label name.
Object <i>params</i>	A list of parameter values for substitution on the server. This parameter can be <code>null</code> if parameter substitution is done on the client.

The `labelExists` method indicates whether the specified label is defined or not. Its method parameters are identical to the first two parameters for `getLabel`.

### See Also:

[Plugging in Custom Code with Adapters](#)

[Input Component Labels](#)

[Dynamically Populating Label Parameters](#)

## 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"/>
    <ui:button label="Set Label" aura:id="button1" press="{!c.setLabel}"/>
    <auradocs:inner aura:id="inner" label="{!v._label}"/>
</aura:label>

```

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"/>
    <ui:inputTextarea aura:id="textarea"
                     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](#)

# Chapter 7

## Supporting Accessibility

### In this chapter ...

- [Accessibility Considerations](#)
- [Buttons](#)
- [Carousels](#)
- [Help and Error Messages](#)
- [Forms, Fields, and Labels](#)
- [Images](#)
- [Events](#)

Aura components are created with accessibility in mind. A sample of components and that are built based on the [WAI-ARIA 1.0 Authoring Practices](#) are as follows. This guideline also applies to components that extend these components.

- `ui:autocomplete` — for autocompleting dropdowns
- `ui:carousel` — for carousel interactions
- `ui:datePicker` — for calendar pickers
- `ui:dialog` — for modal and non-modal overlays
- `ui:image` — for images and icons
- `ui:input` — for input elements such as text fields and date fields
- `ui:menu` — for menus, dropdowns, and muttons

When customizing these components or their sub-components, be careful in preserving code that ensures accessibility, such as the `aria` attributes. See [User Interface Overview](#) for components you can use in your apps.

### Accessibility Testing

Testing your app for accessibility ensures that the app is accessible by users with disabilities or those who use assistive technologies. For apps developed on Android devices, use [Talkback](#) and [Explore by Touch](#). For iOS devices, use [VoiceOver](#) to test for accessibility. For desktop devices, use [Wave Toolbar](#) to check for accessibility errors and recommendations for your apps.

To check that a component's HTML output is accessibility compliant, run `$A.test.assertAccessible()`. You can also run `$A.devToolService.checkAccessibility()` on a browser console.

### Accessibility Anti-Patterns

When testing or developing with Aura and HTML markup, avoid:

- Images without the `alt` attribute
- Anchor element without textual content
- `input` elements without an associated label
- Group of radio buttons not in a `fieldset` tag
- `iframe` or `frame` elements with empty `title` attribute
- `fieldset` element without a `legend`
- `th` element without a `scope` attribute
- `head` element with an empty `title` attribute
- Headings (`H1`, `H2`, etc.) increasing by more than one level at a time
- CSS color contrast ratio between text and background less than 4.5:1

## Accessibility Considerations

Accessible software and assistive technology enable users with disabilities to use and interact with the products you build. We recommend that you follow the [WCAG Guidelines](#) for accessibility when developing with Aura. This guide explains the accessibility features of Aura components in the `ui` namespace, which you can take advantage of while using these components.

### See Also:

[Supporting Accessibility](#)

[Components](#)

[Handling Events with Client-Side Controllers](#)

## Buttons

Buttons are sometimes designed to appear with just an image, without any text. To create an accessible button, use `ui:button` and set a textual label using the `label` attribute. To hide the label from view, set `labelDisplay="false"`. The text is available to assistive technologies, but not be visible on screen.



```
<ui:button label="Search" iconImgSrc="/auraFW/resources/aura/images/search.png"/>
```



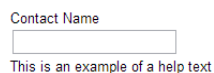
```
<ui:button label="Search" iconImgSrc="/auraFW/resources/aura/images/search.png"
labelDisplay="false"/>
```

## Carousels

The `ui:carousel` component displays a list of items horizontally where users can swipe through the list or click through the page indicators. Note that the carousel will not be accessible if `visible="false"` is set on the `ui:carouselPageIndicatorItem`, since this setting hides the page indicators from view. Similarly, setting `continuousFlow="true"` on `ui:carousel` hides the page indicators from view.

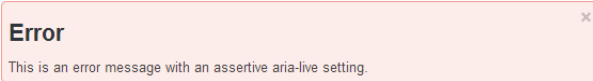
## Help and Error Messages

Use the `ariaDescribedby` attribute to associate the help text or error message with a particular field.



```
<ui:inputText label="Contact Name" labelPosition="top" ariaDescribedby="contact" />
<ui:outputText aura:id="contact" value="This is an example of a help text." />
```

To convey audio notifications, use the `ui:message` component, which has `aria-live="assertive"` and `role="alert"` set on the component by default.



```
<ui:message title="Error" severity="error" closable="true">
  This is an error message.
</ui:message>
```

### See Also:

[Validating Fields](#)

## Forms, Fields, and Labels

Aura's 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. You can assign a label in two ways. Use the `label` attribute on a component that extends `ui:input` or use the `ui:label` component and bind it to the corresponding input component. When using a placeholder in an input component, set the `label` attribute for accessibility.

Use the input components that extend `ui:input`, except when `type="file"`. For example, use `ui:inputTextarea` in preference to the `<textarea>` tag for multi-line text input or the `ui:inputSelect` component in preference to the `<select>` tag.

```
<ui:inputText label="Search" labelPosition="hidden" placeholder="Search" />
```

Designs often include form elements with placeholder text, but no visible label. A label is required for accessibility and can be hidden visually. Set `labelDisplay="false"` to hide it from view but make the component accessible.

```
<ui:label labelDisplay="false" for="myInput" label="My Input Text" />
<ui:inputText aura:id="myInput" value="Put your input here." />
```

### See Also:

[Using Labels](#)

## Images

For an image to be accessible, set an appropriate alternative text attribute. If your image is informational, or actionable as part of a hyperlink, set the `alt` attribute to a descriptive alternative text. If the image is purely decorative, set `imageType="decorative"`. This generates a null `alt` attribute in the `img` tag.

```
<ui:image src="s.gif" imageType="informational" alt="Open Menu">
<ui:image src="s.gif" imageType="decorative">
```

When displaying an informational or actionable image via CSS, include the `assistiveText` class to provide an appropriate alternative text.

```
<a class="like">
  <span class="assistiveText">Like</span>
</a>
```

### Using Images

## Using Images

To display images, use the `ui:image` component. `ui:image` automates common usages of the HTML `<img>` tag, such as `href` linking and other attributes. For an example on how you can use component attributes in images to switch between CSS classes, take a look at the `ui:outputCheckbox` component.

Additionally, include the `imageType` attribute to denote if the image is informational or decorative. Use the `title` attribute for tooltips, especially for icons.

### Informational Images

Informational images can provide information that may not be available in the text, such as a Like or Follow image. They are actionable and can stand alone in a button or hyperlink. Include the `alt` tag to specify an alternate text for the image, which is helpful if the user has no access to the image.

```
<ui:image src="follow.png" imageType="informational" alt="follow" />
```

If you use CSS to display an informational image, you must provide assistive text that will be put into the DOM, by using the `assistiveText` class.

```
<div class="Following">
  <span class="assistiveText">Following</span>
</div>
```

If you use an icon font to display an informational image, provide assistive text that will be put into the DOM.

```
<a class="icon-like">
  <span class="assistiveText">Like</span>
</a>
```

### Decorative Images

Decorative Images are images that can be removed without affecting the logic or content of the page. You don't need to specify assistive text for decorative images.

```
<ui:image src="decoration.png" imageType="decorative" />
```

### See Also:

[Accessibility Considerations](#)

## 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>`, `<input>` in an Aura component. They should not be used on `<div>` and `<span>` tags for accessibility. For a non-actionable element, wrap `<a>` around it and add `onclick`.

```
<a onclick="{!c.doSomething}">{!v.body}</a>
```



# COMMUNICATING WITH EVENTS

## Chapter 8

### Events

---

#### In this chapter ...

- [Handling Events with Client-Side Controllers](#)
- [Component Events](#)
- [Application Events](#)
- [Event Handling Lifecycle](#)
- [Advanced Events Example](#)
- [Firing Aura Events from Non-Aura Code](#)
- [Events Best Practices](#)
- [Events Fired During the Rendering Lifecycle](#)

If you have ever developed with JavaScript or Java Swing, you should be familiar with the idea of 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.

Aura 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.

Aura events are declared by the `aura:event` tag in a `.evt` resource, and they can have one of two types: component or application. Application and component events are declared in separate files, for example, `drawApp/pickBrushComp/pickBrushComp.evt` and `drawApp/pickBrushApp/pickBrushApp.evt`. The event type is set by either `type="COMPONENT"` or `type="APPLICATION"` in the `aura:event` tag.

## Handling Events with Client-Side Controllers

A client-side controller handles events within a component. It's a JavaScript file that defines the functions for all of the component's actions.

Each action function takes in three parameters: the component to which the controller belongs, the event that the action is handling, and the helper if it's used.

### Creating a Client-Side Controller

A client-side controller is part of the component bundle. It is auto-wired if you follow the naming convention, `<componentName>Controller.js`.

To reuse a client-side controller from another component, use the `controller system` attribute in `aura:component`. For example, this component uses the auto-wired client-side controller for `auradocs.sampleComponent` in `auradocs/sampleComponent/sampleComponentController.js`.

```
<aura:component
  controller="js://auradocs.sampleComponent">
  ...
</aura:component>
```

### Calling Client-Side Controller Actions

Let's start by looking at events on an HTML tag.

#### 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/>
  <input type="button" value="Auraized HTML Button" onclick="{!c.handleClick}"/>
  <br/>
  <ui:button label="Native Aura Button" press="{!c.handleClick}"/>
  <br/>
  {!v.text}
</aura:component>
```

#### Client-side controller source

```
{
  handleClick : function(component, event) {
    var attributeValue = component.get("v.text");
    aura.log("current text: " + attributeValue);

    var target;
    if (event.getSource) {
      // handling an Aura event
      target = event.getSource(); // this is an Aura Component object
      component.set("v.text", target.get("v.label"));
    } else {
      // handling a native browser event
      target = event.target.value; // this is a DOM element
      component.set("v.text", event.target.value);
    }
  }
}
```

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. Arbitrary JavaScript in the component is ignored.

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 Aura. However, the "Flawed" button won't work with Aura. The reason is that Aura has its own event system. DOM events are mapped to Aura events, since HTML tags are mapped to Aura components.

## Handling Aura Events

Handle Aura events using actions in client-side component controllers. Aura events for common mouse and keyboard interactions are available with components that come out-of-the-box with Aura. When you extend these components, you have access to these events as well. For example, if you extend the `ui:input` component, you have access to its events, such as `mouseover`, `cut`, and `copy`.

Let's look at the `onclick` attribute in the "Auraized" button, which invokes the `handleClick` action in the controller. The "Native" button uses the same syntax with the `press` attribute in the `<ui:button>` component.

In this simple scenario, there is little functional difference between working with the "Native" button or the "Auraized" HTML button. However, Aura 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 components that come out-of-the-box with Aura 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.

## 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.

`component.get("v.<attributeName>")` returns the value of the `<attributeName>` attribute. The `aura.log()` utility function attempts to find a browser console and logs the attribute value to it.

## Invoking Another Action in the Controller

To call an action method from another method, use a helper function and invoke it using `helper.someFunction(component)`. A helper resource contains functions that can be reused by your JavaScript code in the component bundle.

### See Also:

[Sharing JavaScript Code in a Component Bundle](#)

[Event Handling Lifecycle](#)

[Invoking Actions on Component Initialization](#)

[Creating Server-Side Logic with Controllers](#)

## Component Events

A component event can be handled by a component itself or by a component that instantiates or contains the component.

### Create Custom Component Event

You can create custom component events 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 is a component event with one message attribute.

```
<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 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.

## Register Component Event

A component registers that it may fire an event by using `<aura:registerEvent>` in its markup. For example:

```
<aura:registerEvent name="sampleComponentEvent" type="auradocs:compEvent"/>
```

We'll see how the value of the `name` attribute is used for firing and handling events.

## Fire Component Event

To get a reference to a component event in JavaScript, use `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");
// set some data for the event (also known as event shape)
// compEvent.setParams(...);
compEvent.fire();
```

## 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="auradocs:compEvent"/>
<aura:handler name="sampleComponentEvent" action="{!c.handleSampleEvent}"/>
```



**Note:** The `name` attributes in `<aura:registerEvent>` and `<aura:handler>` must match.

## Handle Component Event of Instantiated Component

The component that registers an event declares the `name` attribute of the event. For example, an `<auradocs:eventsNotifier>` component contains a `<aura:registerEvent>` tag.

```
<aura:registerEvent name="sampleComponentEvent" type="auradocs:compEvent"/>
```

When you instantiate `<auradocs:eventsNotifier>` in another component, use the value of the `name` attribute from the `<aura:registerEvent>` tag to register the handler. For example, if an `<auradocs:eventsHandler>` component includes

`<auradocs:eventsNotifier>` in its markup, `eventsHandler` instantiates `eventsNotifier` and can handle any events thrown by `eventsNotifier`. Here's how `<auradocs:eventsHandler>` instantiates `<auradocs:eventsNotifier>`:

```
<auradocs:eventsNotifier sampleComponentEvent="{!c.handleComponentEventFired}"/>
```

Note how `sampleComponentEvent` matches the value of the `name` attribute in the `<aura:registerEvent>` tag in `<auradocs:eventsNotifier>`.

## Handle Component Event in a Child Component

A child component that extends another component can also handle events fired by the super component. The child component automatically inherits event handlers from its super component.

## Handle Component Event 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 [Dynamically Adding Event Handlers](#) on page 119.

## Get the Source of a Component Event

Use `evt.getSource()` in JavaScript to find out which component fired the component event, where `evt` is a reference to the event.

### See Also:

[Application Events](#)

[Handling Events with Client-Side Controllers](#)

[Advanced Events Example](#)

[What is Inherited?](#)

## 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.

The event and components in this example are in a `docsample` namespace. There is nothing special about this namespace but it's referenced in the code in a few places. Change the code to use a different namespace if you prefer.

### Component Event

#### `ceEvent.evt`

This component event has one attribute. We'll use this attribute to pass some data in the event when it's fired.

```
<aura:event type="COMPONENT">
  <aura:attribute name="message" type="String"/>
</aura:event>
```

### Notifier Component

#### `ceNotifier.cmp`

The component uses `aura:registerEvent` to declare that it may fire the component event.

The button in the component contains a `press` browser event that is wired to the `fireComponentEvent` action in the client-side controller. The action is invoked when you click the button.

```
<aura:component>
  <aura:registerEvent name="cmpEvent" type="docsample:ceEvent"/>

  <h1>Simple Component Event Sample</h1>
  <p><ui:button
    label="Click here to fire a component event"
    press="{!c.fireComponentEvent}" />
  </p>
</aura:component>
```

### ceNotifierController.js

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.

```
{
  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

#### ceHandler.cmp

The handler component contains the `<docsample:ceNotifier>` component and uses the value of the `name` attribute, `cmpEvent`, from the `<aura:registerEvent>` tag in `<docsample:ceNotifier>` to register the handler.

When the event is fired, the `handleComponentEvent` action in the client-side controller of the handler component is invoked.

```
<aura:component>
  <aura:attribute name="messageFromEvent" type="String"/>
  <aura:attribute name="numEvents" type="Integer" default="0"/>

  <!-- handler contains the notifier component
  Note that cmpEvent is the value of the name attribute in aura:registerEvent
  in ceNotifier.cmp -->
  <docsample:ceNotifier cmpEvent="{!c.handleComponentEvent}"/>

  <p>{!v.messageFromEvent}</p>
  <p>Number of events: {!v.numEvents}</p>
</aura:component>
```

### ceHandlerController.js

The controller retrieves the data sent in the event and uses it to update the `messageFromEvent` attribute in the handler component.

```
{
  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

You can test this code by adding the resources to a sample application and navigating to the handler component. For example, if you have a `docsample` application, navigate to:

`http://localhost:<port>/docsample/ceHandler.cmp`.

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.

### Create Custom Application Event

You can create custom application events 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 is an application event with one `message` attribute.

```
<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 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.

## Register Application Event

A component registers that it may fire an application event by using `<aura:registerEvent>` in its markup. Note that the `name` attribute is required but not used for application events. The `name` attribute is only relevant for component events.

```
<aura:registerEvent name="appEvent" type="auradocs:appEvent"/>
```

## Fire Application Event

Use `$A.get("e.myNamespace:myAppEvent")` in JavaScript to get an instance of the `myAppEvent` event in the `myNamespace` namespace. Use `fire()` to fire the event.

```
var appEvent = $A.get("e.auradocs:appEvent");
// set some data for the event (also known as event shape)
//appEvent.setParams({ ... });
appEvent.fire();
```

## Handle Application Event

Use `<aura:handler>` in the markup of the handler component. The `action` attribute of `<aura:handler>` sets the client-side controller action to handle the event. For example:

```
<aura:handler event="auradocs:appEvent" action="{!c.handleApplicationEvent}"/>
```

When the event is fired, the `handleApplicationEvent` client-side controller action is called.

## Get the Source of an Application Event

Note that `evt.getSource()` doesn't work for application events. It only works for component events. A component event is usually fired by code like `cmp.getEvent('myEvt').fire()`; so it's obvious who fired the event. However, it's relatively opaque which component fired an application event. It's fired by code like `$A.getEvt('myEvt').fire()`. If you need to find the source of an application event, you could use `evt.setParams()` to set the source component in the event data before firing it. For example, `evt.setParams("source" : sourceCmp)`, where `sourceCmp` is a reference to the source component.

## Events Fired on Application Rendering

Several events are fired when an Aura application is rendering. All `init` events are fired to indicate the component or application has been initialized. If a component is contained in another component or application, the inner component is initialized first. If any server calls are made during rendering, `aura:waiting` is fired. Finally, `aura:doneWaiting` and `aura:doneRendering` are fired in that order to indicate that all rendering has been completed.

### See Also:

[Component Events](#)

[Handling Events with Client-Side Controllers](#)

[Advanced Events Example](#)

[What is Inherited?](#)

## 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.

The event and components in this example are in a `docsample` namespace. There is nothing special about this namespace but it's referenced in the code in a few places. Change the code to use a different namespace if you prefer.

## Application Event

### `aeEvent.evt`

This application event has one attribute. We'll use this attribute to pass some data in the event when it's fired.

```
<aura:event type="APPLICATION">
  <aura:attribute name="message" type="String"/>
</aura:event>
```

## Notifier Component

### `aeNotifier.cmp`

The notifier component uses `aura:registerEvent` to declare that it may fire the application event. Note that 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 `press` browser event that is wired to the `fireApplicationEvent` action in the client-side controller. Clicking this button invokes the action.

```
<aura:component>
  <aura:registerEvent name="appEvent" type="docsample:aeEvent"/>

  <h1>Simple Application Event Sample</h1>
  <p><ui:button
    label="Click here to fire an application event"
    press="{!c.fireApplicationEvent}" />
  </p>
</aura:component>
```

### `aeNotifierController.js`

The client-side controller gets an instance of the event by calling `$A.get("e.docsample:aeEvent")`. The controller sets the `message` attribute of the event and fires the event.

```
{
  fireApplicationEvent : function(cmp, event) {
    // Get the application event by using the
    // e.<namespace>.<event> syntax
    var appEvent = $A.get("e.docsample:aeEvent");
    appEvent.setParams({
      "message" : "An application event fired me. " +
        "It all happened so fast. Now, I'm everywhere!" });
    appEvent.fire();
  }
}
```

## Handler Component

### `aeHandler.cmp`

The handler component uses the `<aura:handler>` tag to register that it handles the application event.

When the event is fired, the `handleApplicationEvent` action in the client-side controller of the handler component is invoked.

```
<aura:component>
  <aura:attribute name="messageFromEvent" type="String"/>
  <aura:attribute name="numEvents" type="Integer" default="0"/>

  <aura:handler event="docsample:aeEvent" action="{!c.handleApplicationEvent}"/>

  <p>{!v.messageFromEvent}</p>
  <p>Number of events: {!v.numEvents}</p>
</aura:component>
```

### aeHandlerController.js

The controller retrieves the data sent in the event and uses it to update the `messageFromEvent` attribute in the handler component.

```
{
  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

#### aeContainer.cmp

The container component contains the notifier and handler components. This is different from the component event example where the handler contains the notifier component.

```
<aura:component>
  <docsample:aeNotifier/>
  <docsample:aeHandler/>
</aura:component>
```

### Put It All Together

You can test this code by adding the resources to a sample application and navigating to the container component. For example, if you have a `docsample` application, navigate to:

`http://localhost:<port>/docsample/aeContainer.cmp`.

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:

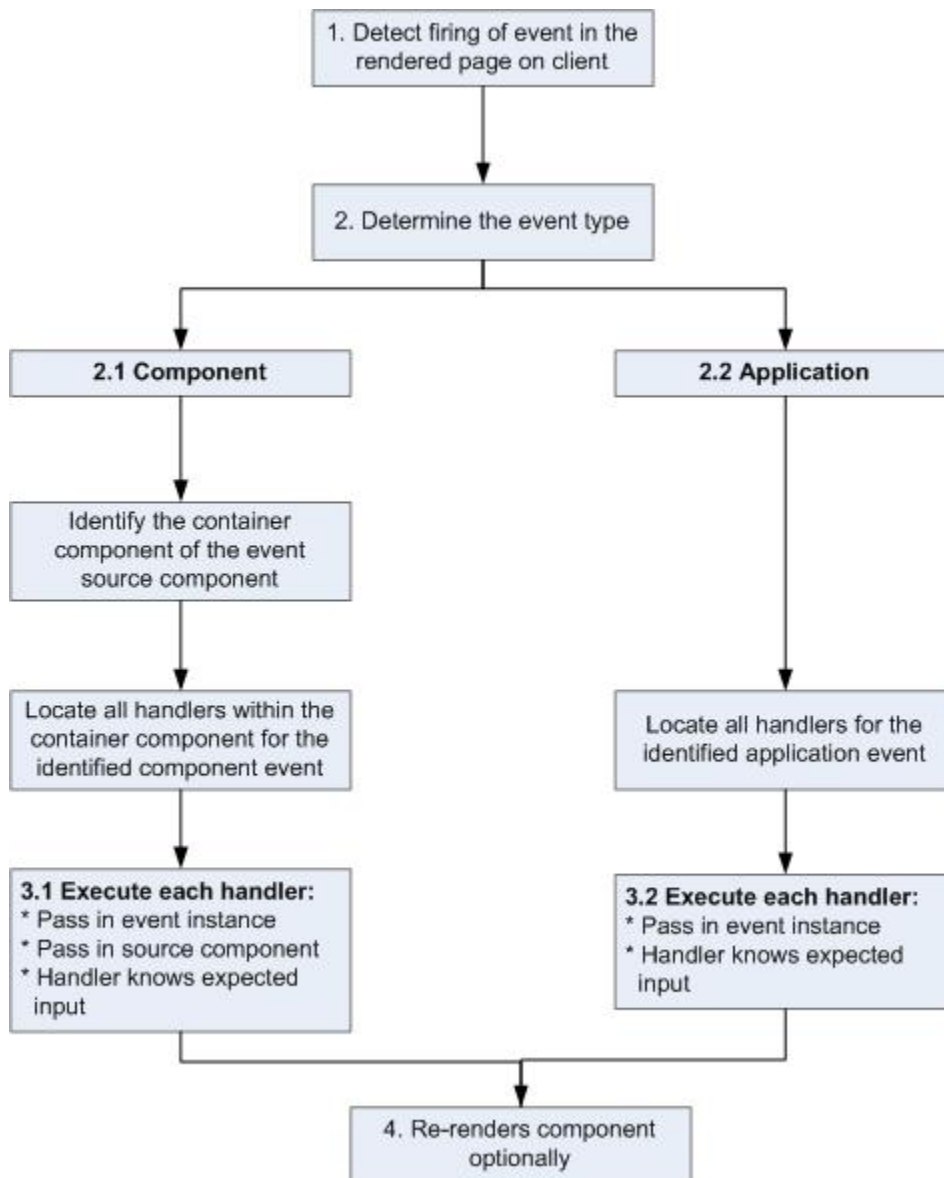
[Application Events](#)

[Creating Server-Side Logic with Controllers](#)

[Component Event Example](#)

## Event Handling Lifecycle

The following chart summarizes how Aura handles events.



### 1 Detect Firing of Event

Aura 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:

[Client-Side Rendering to the DOM](#)

## 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 ( <code>compEvent.evt</code> ) and application event ( <code>appEvent.evt</code> )	Defines the component and application events in separate resources. <code>eventsContainer.cmp</code> shows how to use both component and application events.
Notifier	Component ( <code>eventsNotifier.cmp</code> ) and its controller ( <code>eventsNotifierController.js</code> )	The notifier contains an <code>onclick</code> browser event to initiate the event. The controller fires the event.
Handler	Component ( <code>eventsHandler.cmp</code> ) and its controller ( <code>eventsHandlerController.js</code> )	The handler component contains the notifier component (or a <code>&lt;aura:handler&gt;</code> tag for application events), and calls the controller action that is executed after the event is fired.
Container Component	<code>eventsContainer.cmp</code>	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

### `compEvent.evt`

```
<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

### `appEvent.evt`

```
<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

### `eventsNotifier.cmp`

The notifier component contains a `press` browser event 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 left empty for the application event.

The `parentName` attribute is not set yet. We will see how this attribute is set and surfaced in `eventsContainer.cmp`.

### Component source

```
<aura:component>
  <aura:attribute name="parentName" type="String"/>
  <aura:registerEvent name="componentEventFired" type="auradocs:compEvent"/>
  <aura:registerEvent name="appEvent" type="auradocs:appEvent"/>

  <div>
    <h3>This is {!v.parentName}'s eventsNotifier.cmp instance</h3>
    <p><ui:button
      label="Click here to fire a component event"
      press="{!c.fireComponentEvent}" />
    </p>
    <p><ui:button
      label="Click here to fire an application event"
      press="{!c.fireApplicationEvent}" />
    </p>
  </div>
</aura:component>
```

### CSS source

```
.auradocsEventsNotifier {
  display: block;
  margin: 10px;
  padding: 10px;
  border: 1px solid black;
}
```

### Client-side controller source

The controller fires the event.

```
{
  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.auradocs: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

### eventsHandler.cmp

The handler component contains the notifier component or a `<aura:handler>` tag, and calls the controller action that is executed after the event is fired.

### Component source

```
<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="auradocs:appEvent" action="{!c.handleApplicationEventFired}"/>

  <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>
    <auradocs:eventsNotifier parentName="{!v.name}"
    componentEventFired="{!c.handleComponentEventFired}"/>
  </div>
</aura:component>
```

### CSS source

```
.auradocsEventsHandler {
  display: block;
  margin: 10px;
  padding: 10px;
```

```
border: 1px solid black;
}
```

### Client-side controller source

```
{
  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

### eventsContainer.cmp

#### Component source

```
<aura:component>
  <auradocs:eventsHandler name="eventsHandler1"/>
  <auradocs: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.

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 Aura Events from Non-Aura Code

You can fire an Aura event 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. Let's call this event `mynamespace:externalEvent`. You'll fire this event when your non-Aura code is done by including this JavaScript in your non-Aura code.

```
var myExternalEvent;  
if(window.opener.$A &&  
    (myExternalEvent = window.opener.$A.get("e.mynamespace:externalEvent"))) {  
    myExternalEvent.setParams({isOauthed:true});  
    myExternalEvent.fire();  
}
```

`window.opener.$A.get()` references the master window where your Aura app is loaded.

### See Also:

[Application Events](#)

[Modifying Components from External JavaScript](#)

## Events Best Practices

Here are some best practices for working with events.

### Separate Low-Level Events from Business Logic Events

It's a good practice to 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

If you need 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 to determine the next action to take.
3. If you need to reuse the logic in your component bundle, put the logic in the helper.

For example:

1. Your component markup contains `<ui:button label="do something" press="{!c.click}" />`.
2. In your controller, define the `click` 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, it may be better to identify a dispatcher component to listen for the event. The dispatcher component can perform some logic to decide which component instances should 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 116.

### Don't Use `onclick` and `ontouchend` Events

You can't use different actions for `onclick` and `ontouchend` events in a component. Aura translates touch-tap events into clicks and activates any `onclick` handlers that are present.

### See Also:

[Client-Side Rendering to the DOM](#)  
[Events Best Practices](#)

## Events Fired During the Rendering Lifecycle

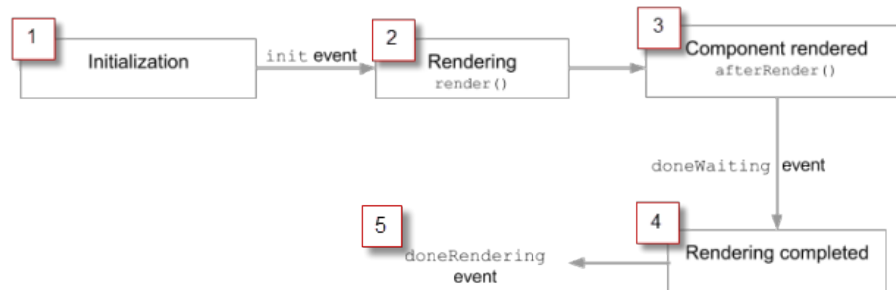
A component is instantiated, rendered, and rerendered during its lifecycle. A component is rerendered only when there's a programmatic or value change that would require a rerender, such as when a browser event triggers an action that updates its data.

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.

Before going into the rendering lifecycle on the client, it's useful to understand the server-side and client-side processing for component requests in brief. Aura builds the component definition and all its dependencies in the server, including definitions for interfaces, controllers, actions, and models. After creating a component instance, the serialized component definitions and instances are sent 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 used to render the component instance. The client locates the custom renderer in the component bundle or uses the default renderer method.

The following image depicts a typical rendering lifecycle of a component on the client, after the component definitions and instances are deserialized.



1. The `init` event is fired by the component service that constructs the components 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. For more information, see [Invoking Actions on Component Initialization](#) on page 116.

2. `render()` is called to start component rendering. The renderer for `aura:component` has a base implementation of `render()`, but your component can override this method in a custom renderer. For more information, see [Client-Side Rendering to the DOM](#) on page 105.
3. `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 Aura rendering service has inserted DOM elements.
4. To indicate that the client is done waiting for a response to the server request XHR, the `doneWaiting` event is fired. You can handle this event by adding a handler wired to a client-side controller action.
5. Aura checks whether any components need to be rerendered and rerenders any “dirty” components to reflect any updates to attribute values, for example. Finally, the `doneRendering` event is fired the end of the rendering lifecycle.

Let's see what happens when a `ui:button` component is returned from the server and any rerendering that occurs when the button is clicked to update its label.

```
<!-- The uiExamples:buttonExample container component -->
<aura:component>
    <aura:attribute name="num" type="Integer" default="0"/>
    <ui:button aura:id="button" label="{!v.num}" press="{!c.update}"/>
</aura:component>
```

```
/** Client-side Controller */
({
    update : function(cmp, evt) {
        cmp.set("v.num", cmp.get("v.num")+1);
    }
})
```



**Note:** It's helpful to refer to the `ui:button` source to understand the component definitions to be rendered. For more information, see <https://github.com/forcedotcom/aura/blob/master/aura-components/src/main/components/ui/button/button.cmp>.

After initialization, `render()` is called to render `ui:button`. `ui:button` doesn't have a custom renderer, and uses the base implementation of `render()`. In this example, `render()` is called eight times in the following order.

Component	Description
<code>uiExamples:buttonExample</code>	The top-level component that contains the <code>ui:button</code> component
<code>ui:button</code>	The <code>ui:button</code> component that's in the top-level component
<code>aura:html</code>	Renders the <code>&lt;button&gt;</code> tag
<code>aura:if</code>	The first <code>aura:if</code> tag in <code>ui:button</code> , which doesn't render anything since the button contains no image
<code>aura:if</code>	The second <code>aura:if</code> tag in <code>ui:button</code>
<code>aura:html</code>	The <code>&lt;span&gt;</code> tag for the button label, nested in the <code>&lt;button&gt;</code> tag
<code>aura:expression</code>	The <code>v.num</code> expression
<code>aura:expression</code>	Empty <code>v.body</code> expression

When rendering is done, this example calls `afterRender()` eight times for these component definitions. The `doneWaiting` event is fired, followed by the `doneRendering` event.

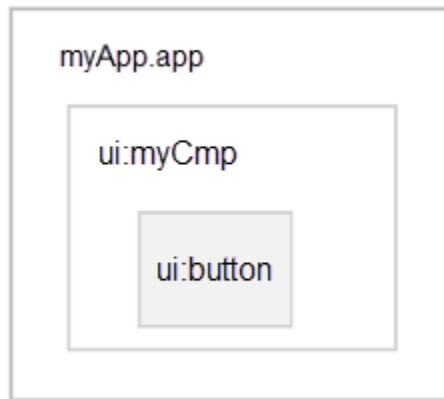
Clicking the button updates its label, which checks for any “dirty” components and fires `rerender()` to rerender these components, followed by the `doneRendering` event. In this example, `rerender()` is called eight times. All changed values are stored in a list on the rendering service, resulting in the rerendering of any “dirty” components.



**Note:** Firing an event in a custom renderer is not recommended. For more information, see [Events Anti-Patterns](#).

## Rendering Nested Components

Let's say that you have an app `myApp.app` that contains a component `ui:myCmp` with a `ui:button` component.



During initialization, the `init()` event is fired in this order: `ui:myCmp`, `ui:button`, and `myApp.app`. The `doneWaiting` event is fired in the same order. Finally, the `doneRendering` event is also called in the same order.

### Customizing the `doneWaiting` Handler

The `doneWaiting` event is fired to signal that the client is done waiting for a response to a server request, and is sometimes preceded by a `waiting` event. The `waiting` event is fired when an action is sent to the server, such as when a server-side action is added using `$A.enqueueAction()` and subsequently run. You can listen for this event by using the following syntax and adding its controller logic.

```
<aura:handler event="aura:waiting" action="{!c.waiting}"/>
<aura:handler event="aura:doneWaiting" action="{!c.doneWaiting}"/>
```

For example, you might want to display a spinner during a `waiting` event and hide it when the `doneWaiting` event is fired. This example either adds or remove a CSS class depending on which event is fired.

```
((
  waiting: function(cmp, event, helper) {
    $A.util.addClass(cmp.find("spinner").getElement(), "waiting");
  },
  doneWaiting: function(cmp, event, helper) {
    $A.util.removeClass(cmp.find("spinner").getElement(), "waiting");
  }
}))
```

### Customizing the `doneRendering` Handler

You can listen for this event by using the following syntax and add its controller logic.

```
<aura:handler event="aura:doneRendering" action="{!c.doneRendering}"/>
```

For example, you want to customize the behavior of your app after it's finished rendering the first time but not after subsequent rerenderings. Create an attribute to determine if it's the first rendering.

```
<aura:attribute name="isDoneRendering" type="Boolean" default="false"/>
```

```
((
  doneRendering: function(cmp, event, helper) {
    if(!cmp.get("v.isDoneRendering")){
```

```
    //do something after app is first rendered
  }
})
```

**See Also:**

[\*Client-Side Rendering to the DOM\*](#)

[\*Server-Side Processing for Component Requests\*](#)

[\*Client-Side Processing for Component Requests\*](#)

#### In this chapter ...

- [App Overview](#)
- [Designing App UI](#)
- [Creating App Templates](#)

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.

## App Overview

An Aura 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 Aura app using the URL scheme:

```
http://<myServer>/<namespace>/<application>.app
```



**Note:** You can access components directly in a browser URL in DEV mode by using the component's `.cmp` extension.

### See Also:

[aura:application](#)

[Supported HTML Tags](#)

## Designing App UI

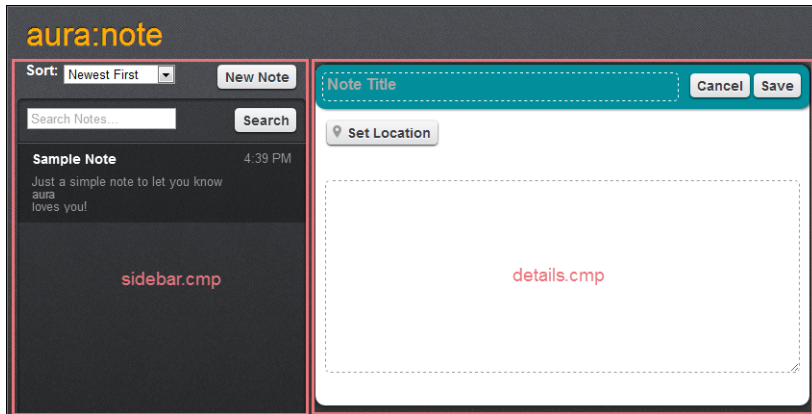
Design your app's UI by including markup in the `.app` resource, which starts with the `<aura:application>` tag.

Let's take a look at the `notes.app` file for the Aura Note sample app.

```
<aura:application template="auranote:template">
  <div>
    <header>
      <h1>aura:note</h1>
    </header>
    <ui:block class="wrapper" aura:id="block">
      <aura:set attribute="left">
        <auranote:sidebar aura:id="sidebar" />
      </aura:set>
      <auranote:details aura:id="details" />
    </ui:block>
  </div>
</aura:application>
```

To learn about system attributes of `<aura:application>`, such as `template`, see [aura:application](#).

`notes.app` contains HTML tags, such as `<h1>` and `<div>`, as well as components, such as `<ui:block>`. We won't go into the details for all the components here but note how simple the markup is. The `<auranote:sidebar>` and `<auranote:details>` components encapsulate the layout for the page.



### See Also:

[aura:application](#)

[Aura Demos](#)

## Creating App Templates

An app template bootstraps the loading of the Aura framework and the app. The default template is `aura:template`.

Customize the default template by creating your own component that extends the default template. For example, the Aura Note sample app has a `auranote:template` template that extends `aura:template`. `auranote:template` looks like:

```
<aura:component isTemplate="true" extends="aura:template">
  <aura:set attribute="title" value="Aura Notes"/>
  ...
</aura:component>
```

Note how the component extends `aura:template` and sets the `title` attribute using `aura:set`. Take a look at the `aura:template` documentation to see the other template attributes that you can customize.

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.

The `notes.app` file points at the custom template by setting the `template` system attribute in `<aura:application>`.

```
<aura:application template="auranote:template">
  ...
</aura:application>
```

### JavaScript Libraries

To use a JavaScript library, you can reference it in your app's template or include a `<script>` tag in the `.app` file. The preferred approach is to add an `<aura:clientLibrary>` tag in a `.cmp` or `.app` resource. See [aura:clientLibrary](#).

To add a JavaScript library to your app's template, use `aura:set` to set the `extraScriptTags` attribute in the template component. This sets the `extraScriptTags` attribute in `aura:template`, which your app's template extends.



For example, the Aura Note sample app uses `ckeditor.js`, which is a third-party JavaScript library. The `auranote:template` includes this markup to include the library.

```
<aura:set attribute="extraScriptTags">
  <script type="text/javascript" src="/aura/ckeditor/ckeditor.js"></script>
</aura:set>
```

You can use multiple `<script>` tags to include more than one library. For example:

```
<aura:set attribute="extraScriptTags">
  <script type="text/javascript" src="/aura/ckeditor/ckeditor.js"></script>
  <script type="text/javascript" src="/aura/codemirror/codemirror.js"></script>
</aura:set>
```

## External CSS

To use an external style sheet, you must link to it in your app's template. Use `aura:set` to set the `extraStyleTags` attribute in the template component. This sets the `extraStyleTags` attribute in `aura:template`, which your app's template extends.

For example:

```
<aura:set attribute="extraStyleTags">
  <link href="/aura/external/google-code-prettify/prettify.css" rel="stylesheet"
type="text/css" />
</aura:set>
```

You can link to multiple external style sheets. For example:

```
<aura:set attribute="extraStyleTags">
  <link href="/aura/external/google-code-prettify/prettify.css" rel="stylesheet"
type="text/css" />
  <link href="/aura/external/morecss/morecss.css" rel="stylesheet" type="text/css"
/>
</aura:set>
```

You can also use inline style in your template, but we recommend using an external style sheet instead. To use inline style, use `aura:set` to set the `inlineStyle` attribute in the template component. For example:

```
<aura:set attribute="inlineStyle">
  <style>
    body {
      background-color: #6cc4e3;
    }
  </style>
</aura:set>
```

## See Also:

[Aura Demos](#)  
[aura:application](#)  
[CSS in Components](#)  
[Using JavaScript Libraries](#)

# Chapter 10

## Styling Apps

---

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`.

### External CSS

Add a `<aura:clientLibrary>` tag in a `.cmp` or `.app` file to specify a CSS library that you want to use. See [aura:clientLibrary](#).

The older method for including external CSS was to add it to your app's template. This method is still supported but `<aura:clientLibrary>` is preferable because it enables you to add the library to the actual component that uses it. Also, it's useful if the location or URL of the library needs to be dynamically generated.

### See Also:

[CSS in Components](#)

[Using JavaScript Libraries](#)

[Aura Demos](#)

[Creating App Templates](#)

## Vendor Prefixes

Vendor prefixes, such as `-moz-` and `-webkit-` among many others, are automatically added in Aura.

You only need to write the unprefixed version, and Aura 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.

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;  
}
```

# Chapter 11

## Adding Components

---

### In this chapter ...

When you're ready to add components to your app, you should first look at the out-of-the-box components that come with Aura. You can also leverage these components by extending them or using composition to add them to custom components that you're building.



**Note:** See the `Components` folder in the [Reference tab](#) for all the components that come out-of-the-box with Aura. The `ui` namespace includes many components that are common on Web pages.

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 should initiate or respond to.

Once you have defined the shape of any new components, developers can work on the components in parallel. This is a useful approach if you have a team working on an app.

### See Also:

- [Component Composition](#)
- [Using Object-Oriented Development](#)
- [Component Attributes](#)
- [Events](#)

# Chapter 12

## Using JavaScript

### In this chapter ...

- [Accessing the DOM](#)
- [Using JavaScript Libraries](#)
- [Working with Attribute Values in JavaScript](#)
- [Working with a Component Body in JavaScript](#)
- [Sharing JavaScript Code in a Component Bundle](#)
- [Client-Side Rendering to the DOM](#)
- [Client-Side Runtime Binding of Components](#)
- [Validating Fields](#)
- [Throwing Errors](#)
- [JavaScript Services](#)

Use JavaScript for client-side Aura code. The `Aura` object is the top-level object in the JavaScript framework code. For all the methods available in the `Aura` class, see the [JavaScript API](#).

You can use `$A` in JavaScript code to denote the `Aura` object; for example, `$A.getCmp()`.

A component bundle can contain JavaScript code in a client-side controller, renderer, helper, or test file. Client-side controllers are the most commonly used of these JavaScript files.

### Publicly Accessible JavaScript Methods

The JavaScript API Reference lists the publicly accessible methods for each JavaScript object. When you are writing code, it's important to understand which methods are publicly accessible for a JavaScript object. Each object has an associated `_export.js` file that lists the public methods that can be called. For example, `ArrayValue.js` has a corresponding `ArrayValue_export.js`. You should only call JavaScript methods that are advertised in an object's `_export.js` file. If you don't, you might get hard-to-debug bugs when you run your code in `PROD` mode.

### 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` files.

## Accessing the DOM

The Document Object Model (DOM) is the language-independent model for representing and interacting with objects in HTML and XML documents. The Aura rendering service takes in-memory component state and updates the component in the DOM.

Aura 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.

There are two very important guidelines for accessing the DOM from Aura.

- You should never modify the DOM outside a renderer. However, you can read from the DOM outside a renderer.
- Use expressions, whenever possible, instead of trying to set a DOM element directly.

### Using Renderers

The rendering service is the bridge from the framework to update the DOM. If you modify the DOM from a client-side controller, the changes may be overwritten when the components are rendered, depending on how the component renderers behave.

### Using Expressions

You can often avoid writing a custom renderer by using expressions in the markup instead.

#### See Also:

[\*Dynamically Showing or Hiding Markup\*](#)

[\*Client-Side Rendering to the DOM\*](#)

[\*Expressions\*](#)

## Using JavaScript Libraries

To use JavaScript libraries in your apps, include `<script>` tags in your `.app` file.

The preferred approach is to add an `<aura:clientLibrary>` tag in a `.cmp` or `.app` resource. See [aura:clientLibrary](#).

The older method for including a JavaScript library was to add it to your app's template. This method is still supported but `<aura:clientLibrary>` is preferable because it enables you to add the library to the actual component that uses it. Also, it's useful if the location or URL of the library needs to be dynamically generated.

#### See Also:

[\*Aura Demos\*](#)

[\*Creating App Templates\*](#)

[\*aura:application\*](#)

## Working with Attribute Values in JavaScript

These are useful and common patterns for working with attribute values 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.

### 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");
```

### Get a Boolean Attribute Value

To get the boolean value of a component's `isCreated` attribute:

```
var isCreated = $A.util.getBooleanValue(cmp.get("v.isCreated"));
```

### 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:

[Accessing Models in JavaScript](#)

[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.

## 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

Remember that the body attribute is an array of components. 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](#)

## 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 firing server-side actions.

They can be called from a client-side controller or renderer for a component.

### Creating a Helper

A helper file is part of the component bundle and is auto-wired if you follow the naming convention, `<componentName>Helper.js`.

To reuse a helper from another component, you can use the `helper` system attribute in `aura:component` instead. For example, this component uses the auto-wired helper for `auradocs.sampleComponent` in `auradocs/sampleComponent/sampleComponentHelper.js`.

```
<aura:component
    helper="js://auradocs.sampleComponent">
    ...
</aura:component>
```



**Note:** If you are reusing a helper from another component and you already have an auto-wired helper in your component bundle, the methods in your auto-wired helper will not be accessible. We recommend that you use a helper within the component bundle for maintainability and use an external helper only if you must.

## 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
    }
})
```

For an example on using helper methods to customize renderers, see [Client-Side Rendering to the DOM](#).

## 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.

The following code shows you how to call the `updateItem` helper function in a controller.

```
((
    newItemEvent: function(component, event, helper) {
        helper.updateItem(component, event.getParams());
    }
})
```



The following code shows the helper function, which takes in the `value` parameter set in the controller via the `item` argument.

```
((
  updateItem : function(component,item) {
    //Get the model
    var items = component.getValue("m.items");
    items.each(function(e, i) {
      //Process updated items here
    });
    //Update the model via a server-side action
    var action = component.get("c.saveItem");
    action.setParams(item);
    //Set any optional callback and enqueue the action
    $A.enqueueAction(action);
  }
))
```

For example, this helper function can be called during both an init event handler and a custom event handler.

### See Also:

[Client-Side Rendering to the DOM](#)

[Component Bundles](#)

[Handling Events with Client-Side Controllers](#)

## Client-Side Rendering to the DOM

The Aura rendering service takes in-memory component state and updates the component in the Document Object Model (DOM).

The DOM is the language-independent model for representing and interacting with objects in HTML and XML documents. Aura 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.

You should never modify the DOM outside a renderer. However, you can read from the DOM outside a renderer.

### Rendering Lifecycle

The rendering lifecycle automatically handles rendering and rerendering of components whenever the underlying data changes. Here is an outline of the rendering lifecycle.

1. A browser event triggers one or more Aura events.
2. Each Aura 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. When all the data updates from the events are processed, the framework rerenders all the components that own modified data.

### Base Component Rendering

The base component in Aura 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 `render()`, `rerender()`, `afterRender()`, and `unrender()` functions. The framework calls these functions as part of the rendering lifecycle. We will learn more about them in this topic. You can override the base rendering functions in a custom renderer.



**Note:** When you create a new component, Aura fires an `init` event, enabling you to update a component or fire an event after component construction but before rendering. The default renderer, `render()`, gets the component body and use the rendering service to render it.

## Creating a Renderer

You don't normally have to write a custom renderer, but if you want to customize rendering behavior, you can create a client-side renderer in a component bundle. 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`.

To reuse a renderer from another component, you can use the `renderer` system attribute in `aura:component` instead. For example, this component uses the auto-wired renderer for `auradocs.sampleComponent` in `auradocs/sampleComponent/sampleComponentRenderer.js`.

```
<aura:component
    renderer="js://auradocs.sampleComponent">
    ...
</aura:component>
```



**Note:** If you are reusing a renderer from another component and you already have an auto-wired renderer in your component bundle, the methods in your auto-wired renderer will not be accessible. We recommend that you use a renderer within the component bundle for maintainability and use an external renderer only if you must.

## Customizing 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 typically 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.



**Note:** These guidelines are very important when you customize rendering.

- A renderer should only modify DOM elements that are part of the component. You should 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.
- A renderer should never fire an event. An alternative is to use an `init` event instead.

## Rerendering 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.

The framework automatically calls `rerender()` if you update data in a component. You only have to explicitly call `rerender()` if you haven't updated the data but you still want to rerender the component.

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.

## Accessing the DOM After Rendering

The `afterRender()` function enables you to interact with the DOM tree after the Aura 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.

If you want to use a library, such as jQuery, to access the DOM, use it in `afterRender()`.

You generally want to extend default after rendering by calling `superAfterRender()` function before you add your custom code.

## Unrendering 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 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.

## Ensuring Client-Side Rendering

Aura calls the default server-side renderer by default, or a client-side renderer if you have one. If you want to ensure client-side rendering of a top-level component, append `render="client"` to the `aura:component` tag. Setting this in the top-level component will take precedence over Aura's detection logic, which takes dependencies into consideration. This is especially useful if you are testing the component directly in your browser and want to inspect the component using the client-side framework when the test loads. Setting `render="client"` for test components ensures that the client-side framework is loaded, even though it normally wouldn't be needed.

## Rendering Example

Let's look at the button component to see how it customizes the base rendering behavior. It is important to know that every tag in Aura, including standard HTML tags, has an underlying Aura component representation. Therefore, the Aura rendering service uses the same process to render standard HTML tags or custom components that you create.

View the source for `ui:button`. Note that the button component includes a `disabled` attribute that enables Aura to track the disabled status for the component in a Boolean.

```
<aura:attribute name="disabled" type="Boolean" default="false"/>
```

In `button.cmp`, `onclick` is set to `{!c.press}`.

The renderer for the button component is `buttonRenderer.js`. The button component overrides the default `render()` function.

```
render : function(cmp, helper) {
    var ret = this.superRender();
    helper.updateDisabled(cmp);
    return ret;
},
```

The first line calls the `superRender()` function to invoke the default rendering behavior. The `helper.updateDisabled(cmp)` call invokes a helper function to customize the rendering.

Let's look at the `updateDisabled(cmp)` function in `buttonHelper.js`.

```
updateDisabled: function(cmp) {
    if (cmp.get("v.disabled")) {
        var disabled = $A.util.getBooleanValue(cmp.get("v.disabled"));
```

```

var button = cmp.find("button");
if (button) {
    var element = button.getElement();
    if (element) {
        if (disabled) {
            element.setAttribute('disabled', 'disabled');
        } else {
            element.removeAttribute('disabled');
        }
    }
}
}
}
}

```

The `updateDisabled(cmp)` function translates the Boolean `disabled` value to the value expected in HTML, where the attribute doesn't exist or is set to `disabled`.

It uses `cmp.find("button")` to retrieve a unique component. Note that `button.cmp` uses `aura:id="button"` to uniquely identify the component. `button.getElement()` returns the DOM element.

The `rerender()` function in `buttonRenderer.js` is very similar to the `render()` function. Note that it also calls `updateDisabled(cmp)`.

```

rerender : function(cmp, helper){
    this.superRerender();
    helper.updateDisabled(cmp);
}

```

Rendering components is part of the lifecycle of the framework and it's a bit trickier to demonstrate than some other concepts in Aura. The takeaway is that you don't need to think about it unless you need to customize the default rendering behavior for a component.

### See Also:

[Accessing the DOM](#)

[Invoking Actions on Component Initialization](#)

[Component Bundles](#)

[Events](#)

[Sharing JavaScript Code in a Component Bundle](#)

[Server-Side Rendering to the DOM](#)

## Client-Side Runtime Binding of Components

A provider enables you to use an abstract component or an interface in markup. The framework uses the provider to determine the concrete component to use at runtime.

Server-side providers are more common, but if you don't need to access the server when you're creating a component, you can use a client-side provider instead.



**Note:** The framework behavior is undefined if a component has a client-side provider and a server-side provider that return different values. It's preferable to only use a server-side or a client-side provider unless you need both.

### Creating a Provider

A client-side provider is part of the component bundle and is auto-wired if you follow the naming convention,

`<componentName>Provider.js`.

To reuse a provider from another component, you can use the `provider` system attribute in `aura:component` instead. For example, this component uses the auto-wired provider for `auradocs.sampleComponent` in `auradocs/sampleComponent/sampleComponentProvider.js`.

```
<aura:component
  provider="js://auradocs.sampleComponent">
  ...
</aura:component>
```



**Note:** If you are reusing a provider from another component and you already have an auto-wired provider in your component bundle, the methods in your auto-wired provider will not be accessible. We recommend that you use a provider within the component bundle for maintainability and use an external provider only if you must.

A client-side provider is a simple JavaScript object that defines the `provide` function. For example, this provider returns a string that defines the topic to display.

```
((
  provide : function (cmp) {
    var topic = cmp.get('v.topic');
    return 'auradocs' + topic + 'Topic';
  }
}))
```

Instead of a string, a provider can return a JSON object to provide both the concrete component and set some additional attributes. For example:

```
((
  provide : function (cmp) {
    var topic = cmp.get('v.topic');
    return {
      componentDef: 'auradocs' + topic + 'Topic',
      attributes: {
        "type": "task"
      }
    }
  }
}))
```

You can omit the `componentDef` entry if the component is already concrete and you only want to provide attributes.

## Declaring Provider Dependencies

The Aura framework automatically tracks dependencies between definitions, such as components. However, if a component uses a provider that instantiates components that are not directly referenced elsewhere, use `<aura:dependency>` in the component markup to explicitly tell the framework about the dependency, which wouldn't otherwise be discovered by Aura.

### See Also:

[Server-Side Runtime Binding of Components](#)

[Abstract Components](#)

[Interfaces](#)

[Component Bundles](#)

[aura:dependency](#)

## Validating Fields

You can validate fields using JavaScript. Typically, you validate the user input, identify any errors, and display the error messages. You can use Aura's default error handling or customize it with your own error handlers.

### Default Error Handling

Aura can handle and display errors using the default error component, `ui:inputDefaultError`, without using custom error handlers. The following example shows how Aura handles a validation error and uses the default error component to display the error message.

#### Component source

```
<aura:component>
    Enter a number: <ui:inputNumber aura:id="inputCmp"/> <br/>
    <ui:button label="Submit" press="{!c.doAction}"/>
</aura:component>
```

#### Client-side controller source

```
{
    doAction : function(component) {
        var inputCmp = component.find("inputCmp");
        var value = inputCmp.get("v.value");

        // is input numeric?
        if (isNaN(value)) {
            // set error
            inputCmp.setValid("v.value", false);
            inputCmp.addErrors("v.value", [{message:"Input not a number: " + value}]);
        } else {
            // clear error
            inputCmp.setValid("v.value", true);
        }
    }
}
```

When you enter a value and click **Submit**, an action in the controller validates the input and displays an error message if the input is not a number. Entering a valid input clears the error. The controller invalidates the input value using `setValid(false)` and clears any error using `setValid(true)`. You can add error messages to the input value using `addErrors()`.

### Custom Error Handling

`ui:input` and its child components can handle errors using its `onError` and `onClearErrors` attributes, which are wired to your custom error handlers defined in a controller. `onError` maps to a `ui:validationError` event, and `onClearErrors` maps to `ui:clearErrors`. The input component can use the `ui:updateError` event to update the default error component, `ui:inputDefaultError`.

The following example shows how you can handle a validation error using custom error handlers and display the error message using the default error component.

#### Component source

```
<aura:component>
    Enter a number: <ui:inputNumber aura:id="inputCmp" onError="{!c.handleError}"
onClearErrors="{!c.handleClearError}"/> <br/>
    <ui:button label="Submit" press="{!c.doAction}"/>
</aura:component>
```

**Client-side controller source**

```

{
  doAction : function(component, event) {
    var inputCmp = component.find("inputCmp");
    var value = inputCmp.get("v.value");

    // is input numeric?
    if (isNaN(value)) {
      // fire event that will set error
      var errorEvent = inputCmp.getEvent("onError");
      errorEvent.setParams({ "errors" : [{message:"Input not a number: " + value}]});

      errorEvent.fire();
    } else {
      // fire event that will clear error
      var clearErrorEvent = inputCmp.getEvent("onClearErrors");
      clearErrorEvent.fire();
    }
  },

  handleError: function(component, event){
    var inputCmp = component.find("inputCmp");
    var errorsObj = event.getParam("errors");

    /* do any custom error handling
     * logic desired here */

    // set error using default error component
    inputCmp.setValid("v.value", false);
    inputCmp.addErrors(errorsObj);
    var updateErrorEvent = inputCmp.getEvent("updateError");
    updateErrorEvent.fire();
  },

  handleClearError: function(component, event) {
    var inputCmp = component.find("inputCmp");

    /* do any custom error handling
     * logic desired here */

    // clear error using default error component
    inputCmp.setValid("v.value", true);
    var updateErrorEvent = inputCmp.getEvent("updateError");
    updateErrorEvent.fire();
  }
}

```

When you enter a value and click **Submit**, an action in the controller executes. However, instead of letting Aura handle the errors, you have to provide a custom error handler using the `onError` attribute in the component. If the validation fails, `doAction` adds an error message using `setParams()` and fires your custom error handler. In the custom event handler, `handleError`, retrieve the errors by calling `getParam()` and invalidate the input value using `setValid(false)`. You can fire the `updateError` event to update the default error component.

Similarly, you can customize how you want to clear the errors by using the `onClearErrors` event. See the `handleClearError` handler in the controller for an example.

**See Also:**

[Handling Events with Client-Side Controllers](#)  
[Component Events](#)

## Throwing Errors

Aura gives you flexibility in handling unrecoverable and recoverable app errors in JavaScript code.

### Unrecoverable Errors

Use `$A.error("error message here")` for unrecoverable errors, such as an error that prevents your app from starting successfully. It shows a stack trace on the page.

### Recoverable Errors

To handle recoverable errors, use a component, such as `ui:message` or `ui:dialog`, to tell the user about the problem.

This sample shows you the basics of throwing and catching an error in a JavaScript controller.

#### Component source

```
<aura:component>
    <br/>
    <p>Click the button to trigger the controller to throw an error.</p>
    <br/>
    <div aura:id="div1"></div>

    <ui:button label="Throw an Error" press="{!c.throwErrorForKicks}" />
</aura:component>
```

#### Client-side controller source

```
{
    throwErrorForKicks: function(cmp) {
        // this sample always throws an error
        var hasPerm = false;
        try {
            if (!hasPerm) {
                throw new Error("You don't have permission to edit this record.");
            }
        }
        catch (e) {
            // config for a dynamic ui:message component
            var componentConfig = {
                componentDef : "markup://ui:message",
                attributes : {
                    values : {
                        title : "Sample Thrown Error",
                        severity : "error",
                        body : [
                            {
                                componentDef : "markup://ui:outputText",
                                attributes : {
                                    values : {
                                        value : e.message
                                    }
                                }
                            }
                        ]
                    }
                }
            };
        }
    }
};
```



```

        $A.componentService.newComponentAsync (
            this,
            function (message) {
                var div1 = cmp.find("div1");

                // Replace existing body with the dynamic component
                div1.set("v.body", message);
            },
            componentConfig
        );
    }
}
))

```

See the controller code for an example of throwing an error in a try-catch block. The message in the error is displayed to the user in a dynamically created `ui:message` component.

### See Also:

[Validating Fields](#)

## JavaScript Services

Aura provides a set of client-side services that helps you develop apps faster. You can call these services from your JavaScript code, using the syntax `$A.<service>.<method>`.

### AuraComponentService (`$A.componentService`)

The component service enables you to create and manage components. You can use the service to create a component on the client or server and access or manipulate component data. For example, use

`$A.componentService.newComponentAsync()` to create a component dynamically. See [Dynamically Creating Components](#) for an example of using the component service.

### AuraDevToolService (`$A.devToolService`)

The dev tool service helps you debug and test in non-production modes by enabling you to query the current state of objects and get statistics about a running app. For example, you can get a list of available views using `$A.devToolService.views`.

### AuraEventService (`$A.eventService`)

The event service enables you to create and manage events. For example, you can create a new application event using `$A.eventService.newEvent("namespace:component")`.

### AuraExpressionService (`$A.expressionService`)

The expression service enables you to process expressions against a different value provider.

### AuraHistoryService (`$A.historyService`)

The history service enables you to manage browser history and change the location. For example, use `$A.historyService.set(location)` to add the location to the history service, and use `$A.historyService.back()` or `$A.historyService.forward()` to load the previous or subsequent URL or layout on the list.

### AuraLayoutService (`$A.layoutService`)

The layout service enables you to manage the layout and navigation of your app.

**AuraLocalizationService (\$A.localizationService)**

The localization service enables you to manage the localization of date and time, and it's used by many of the input and output components like `ui:inputDate`, `ui:inputDateTime`, `ui:outputDate`, `ui:outputDateTime`, and `ui:outputCurrency`. For example, `$A.localizationService.parseDateTime(value, "yyyy-MM-dd");` parses the datetime value in the given format and returns a JavaScript Date object. See [Localization](#) for an example of using the component service.

**AuraRenderingService (\$A.renderingService)**

The rendering service renders components by retrieving and calling their specific renderers. You can override the default rendering behavior in a client-side renderer. For more information, see [Client-Side Rendering to the DOM](#) on page 105.

**AuraStorageService (\$A.storageService)**

The storage service provides a caching infrastructure for data storage on the client. For more information, see [Using Storage Service](#).

**See Also:**

[Using Layouts for Metadata-Driven Navigation](#)  
[Server-Side Rendering to the DOM](#)  
[Caching with Storage Service](#)

# Chapter 13

## JavaScript Cookbook

---

### In this chapter ...

- [Invoking Actions on Component Initialization](#)
- [Detecting Data Changes](#)
- [Finding Components by ID](#)
- [Dynamically Creating Components](#)
- [Dynamically Adding Event Handlers](#)
- [Creating a Document-Level Event Handler](#)
- [Modifying Components from External JavaScript](#)
- [Dynamically Showing or Hiding Markup](#)
- [Adding and Removing Styles](#)

This section includes code snippets and samples that can be used in various JavaScript files.

## Invoking Actions on Component Initialization

You can update a component or fire an event after component construction but before rendering.

### Component source

```
<aura:component>
  <aura:attribute name="setMeOnInit" type="String" default="default value" />

  <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

  <br/>

  <p>This value is set in the controller after the component initializes and before
  rendering.</p>

  <br/>
  <p><b>{!v.setMeOnInit}</b></p>

</aura:component>
```

### Client-side controller source

```
((
  doInit: function(cmp) {
    // Set the value. This is not a very interesting sample as it just sets an attribute

    // but you could fire an event here instead
    cmp.set("v.setMeOnInit", "controller init magic!");
  }
})
```

Let's look at the **Component source** to see how this works. The magic happens in this line.

```
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
```

This registers an `init` event for the component. After the component is initialized, the `doInit` action is called in the component's controller. In this sample, the controller action sets an attribute value, but it could do something more interesting, such as firing an event.



**Note:** You should never fire an event in a renderer so using the `init` event is a good alternative for many scenarios.

Setting `value="{!this}"` marks this as a value event. You should always use this setting for an `init` event.

### See Also:

[Handling Events with Client-Side Controllers](#)

[Client-Side Rendering to the DOM](#)

[Component Attributes](#)

[Detecting Data Changes](#)

## Detecting Data Changes

### Automatically firing an event

You can configure a component to automatically invoke a client-side controller action when a value in one of the component's model or attributes changes. When the value changes, the `valueChange.evt` event is automatically fired. The `valueChange.evt` is an event with `type="VALUE"` that takes in two attributes, `value` and `index`.

### Manually firing an event

In contrast, other component and application events are fired manually by `event.fire()` in client-side controllers.

For example, in the component, define a handler with `name="change"`.

```
<aura:handler name="change" value="{!v.items}" action="{!c.itemsChange}"/>
```

A component can have multiple `<aura:handler name="change">` tags to detect changes to different attributes.

In addition to the `name` attribute, `aura:handler` includes the `value` and `action` attributes.

Attribute Name	Type	Description
value	Object	The value for which you want to detect changes.
action	Object	The client-side controller action that is run when a change is detected.

In the controller, define the action for the handler.

```
((  
  itemsChange: function(cmp, evt) {  
    var v = evt.getParam("value");  
    if (v === cmp.get("v.items")) {  
      //do something  
    }  
  }  
}))
```

When a change occurs to a value that is represented by the `change` handler, Aura handles the firing of the event and rerendering of the component. For examples of detecting data changes, see the `aura:iteration` and `ui:inputSelect` components.

### See Also:

[Invoking Actions on Component Initialization](#)

## Finding Components by ID

You can retrieve a component by its ID in JavaScript code. For example, a component has a local ID of `button1`.

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

You can find the button component by calling `cmp.find("button1")`, where `cmp` is a reference to the button component in your JavaScript code. The `find()` function has one parameter, which is the local ID of a component within the markup.

You can also retrieve a component by its global ID if you already have a value for the component's `globalId` in your code.

```
var comp = $A.getCmp(globalId);
```

### See Also:

[Component IDs](#)

[Value Providers](#)

## Dynamically Creating Components

You can create a component dynamically from your client-side JavaScript code using the `newComponentAsync()` method.



**Note:** The `newComponentAsync()` method replaces the deprecated `newComponent()` and `newComponentDeprecated()` methods.

`$A.componentService.newComponentAsync(callbackScope, callback, config, attributeValueProvider, localCreation, doForce, forceServer)` takes in a required `callback` function that returns your newly created component, and a required `config` object, which provides the component descriptor and attributes. Refer to the JavaScript API reference for a full description of all the arguments.

This sample code creates a new `ui:button` component with the local ID, attaches an event handler to the new button, and appends the button to the body.

```
createButton : function(cmp) {
    $A.componentService.newComponentAsync(
        this,
        function(newButton) {
            //Pass an event handler to the new button
            newButton.addHandler('press', cmp, 'c.someHandler');
            var body = cmp.get("v.body");
            body.push(newButton);
        },
        {
            "componentDef": "markup://ui:button",
            "localId": "myLocalId",
            "attributes": {
                "values": { label: "Submit" }
            }
        }
    );
}
```

`$A.componentService.newComponentAsync()` is equivalent to `$A.newCmpAsync()`.

### Declaring Dependencies

The Aura framework automatically tracks dependencies between definitions, such as components. However, some dependencies aren't easily discoverable by Aura; for example, if you dynamically create a component that is not directly referenced in the component's markup. To tell the framework about such a dynamic dependency, use the `<aura:dependency>` tag. This ensures that the component and its dependencies are sent to the client, when needed.

For more information about usage, see [aura:dependency](#) on page 202.

## Server-Side Dependencies

The `newComponentAsync()` method supports both client-side and server-side component creation. If no server-side dependencies are found, this method is run synchronously. The top-level component determines whether a server request is necessary for component creation.



**Note:** Creating components where the top-level components don't have server dependencies but nested inner components do is not currently supported. For such cases, set `render="server"` on the top-level `aura:component` or `aura:application` tag.

Server-side dependencies include server-side models, renderers, or providers for the component and its super components. Any server-side models for the component and its super components is a server-side dependency. A server-side controller is not a server-side dependency for component creation as controller actions are only called after the component has been created.

A component with server-side dependencies is created on the server, even if it's preloaded. If there are no server dependencies and the definition already exists on the client via preloading or declared dependencies, no server call is made. To force a server request, set the `forceServer` parameter to `true`.

If a component has both a server-side and client-side renderer or provider, the client-side renderer or provider is used.

### See Also:

[aura:component](#)

[Dynamically Adding Event Handlers](#)

## Dynamically Adding Event Handlers

You can dynamically add a handler for an event that a component fires. The component can be created dynamically on the client-side or fetched from the server at runtime.

This sample code adds an event handler to instances of `auradocs:sampleComponent`.

```
addNewHandler : function(cmp, event) {  
    var cmpArr = cmp.find({ instancesOf : "auradocs:sampleComponent" });  
    for (var i = 0; i < cmpArr.length; i++) {  
        var outputCmpArr = cmpArr[i];  
        outputCmpArr.addHandler("someAction", cmp, "c.someAction");  
    }  
}
```

You can also add an event handler to a component that is created dynamically in the callback function of `$A.services.component.newComponentAsync()`. See [Dynamically Creating Components](#) for more information.

`component.addHandler()` adds an event handler to a component. Note that you cannot force a component to start firing events that it doesn't fire. `c.someAction` can be an action in a controller in the component's hierarchy. `someAction` and `cmp` refers to the event name and value provider respectively. `someAction` must match the name attribute value in the `aura:registerEvent` or `aura:handler` tag. Refer to the JavaScript API reference for a full list of methods and arguments.

### See Also:

[Handling Events with Client-Side Controllers](#)

[Creating Server-Side Logic with Controllers](#)

[Client-Side Rendering to the DOM](#)

## Creating a Document-Level Event Handler

To create a document-level event handler, call `addDocumentLevelHandler(String eventName, Function callback, Boolean autoEnable)`. This creates and returns a handler object that can be enabled and disabled with `setEnabled(Boolean)`.



**Note:** Document-level event handlers are global objects so using many of them could have performance implications.

An example of when a document-level event handler can be useful is with modal dialogs that should close when someone clicks outside of them. Here is an example of how to add a document-level event handler. This code is from the `datePickerHelper.js` code that is part of the `datePicker` component:

```
updateGlobalEventListeners: function(component) {
  var concreteCmp = component.getConcreteComponent();
  var visible = concreteCmp.get("v.visible");
  if (!concreteCmp._clickStart) {
    concreteCmp._clickStart = concreteCmp.addDocumentLevelHandler(
      this.getOnClickEventProp("onClickStartEvent"),
      this.getOnClickStartFunction(component),
      visible);
    concreteCmp._clickEnd = concreteCmp.addDocumentLevelHandler(
      this.getOnClickEventProp("onClickEndEvent"),
      this.getOnClickEndFunction(component),
      visible);
  } else {
    concreteCmp._clickStart.setEnabled(visible);
    concreteCmp._clickEnd.setEnabled(visible);
  }
},
```

The document-level event handlers will be cleaned up automatically when the component is destroyed. If you need to destroy the document-level event handler earlier, call `removeDocumentLevelHandler()`.

## Modifying Components from External JavaScript

You can modify component state outside an event handler and trigger re-rendering of the component. This is particularly useful if you use `window.setTimeout()` in your event handlers to execute some logic after a time delay.

```
window.setTimeout(function () {
  $A.run(function() {
    cmp.set("v.visible", true);
  });
}, 5000);
```

This code sets the `visible` attribute on a component to `true` after a five-second delay. Use `$A.run()` to modify a component outside an event handler and trigger re-rendering of the component by the framework.

### See Also:

[Handling Events with Client-Side Controllers](#)  
[Firing Aura Events from Non-Aura Code](#)  
[Events](#)



## Dynamically Showing or Hiding Markup

You can show or hide markup when a button is pressed.

### Component source

```
<aura:component>
  <aura:attribute name="visible" type="Boolean" default="false" />

  <br/>

  <p>Click the button to see toggling at its best!</p>

  <br/>

  <aura:renderIf isTrue="{!v.visible}">
    <p>Now, you see me!</p>
  </aura:renderIf>

  <ui:button label="Toggle Markup Visibility" press="{!c.showHide}" />

</aura:component>
```

### Client-side controller source

```
((
  showHide: function(cmp) {
    var isVisible = $A.util.getBooleanValue(cmp.get("v.visible"));
    // toggle the visible value
    cmp.set("v.visible", !isVisible);
  }
}))
```

Let's look at the **Component source** to see how this works. We added an attribute called `visible` to control whether the markup is visible. It's set to `false` by default so that the markup is not visible. Under the covers, there are no DOM elements created for the markup.

The `aura:renderIf` tag selectively display the markup in its body if the `visible` attribute evaluates to `true`.

The `ui:button` triggers the `showHide` action in the client-side controller. It simply toggles the value of the `visible` attribute.

### See Also:

[Handling Events with Client-Side Controllers](#)

[Component Attributes](#)

[aura:renderIf](#)

## Adding and Removing Styles

You can add or remove a CSS style to an element during runtime.

The following demo shows how to append and remove a CSS style from an element.

### Component source

```
<aura:component>
  <div aura:id="changeIt">Change Me!</div><br />
  <ui:button press="{!c.applyCSS}" label="Add Style" />
  <ui:button press="{!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 el = cmp.find('changeIt');
    $A.util.addClass(el.getElement(), 'changeMe');
  },

  removeCSS: function(cmp, event) {
    var el = cmp.find('changeIt');
    $A.util.removeClass(el.getElement(), 'changeMe');
  }
}
```

The buttons in this demo are wired to controller actions that append or remove the CSS styles. To append a CSS style to an element, use `$A.util.addClass(element, 'class');`. Similarly, remove the class by using `$A.util.removeClass(element, 'class');` in your controller. `cmp.find()` locates the element using the local ID, denoted by `aura:id="changeIt"` in this demo.

To toggle the class, use `$A.util.toggleClass(element, 'class');`. Refer to the JavaScript API Reference for more utility functions for working with DOM elements.

### See Also:

[Handling Events with Client-Side Controllers](#)

[CSS in Components](#)

[Component Bundles](#)

# Chapter 14

## Using Java

---

### In this chapter ...

- [Essential Terminology](#)
- [Reading Initial Component Data with Models](#)
- [Creating Server-Side Logic with Controllers](#)
- [Server-Side Rendering to the DOM](#)
- [Server-Side Runtime Binding of Components](#)
- [Serializing Exceptions](#)

Use Java to write server-side Aura code. Services are the API in front of Aura. The `Aura` class is the entry point in Java for accessing server-side services.

Your app can contain the following types of Java files.

- Models for initializing component data
- Server-side controllers for handling requests from client-side controllers
- Server-Side Providers for returning a concrete component at runtime for an abstract component or an interface in markup

### See Also:

[Java Models](#)

[Creating Server-Side Logic with Controllers](#)

[Server-Side Runtime Binding of Components](#)

[Component Request Lifecycle](#)

[Using Object-Oriented Development](#)

## Essential Terminology

When you write Java code in Aura, it's essential to understand some basic concepts of the framework.

Term	Description
Definition	<p>Each definition describes metadata for an element, such as a component, event, controller, or model. A large part of Aura is a registry of definitions for its various elements.</p> <p>A definition's metadata can include a name, location of origin, and descriptor (DefDescriptor, the primary key of the definition).</p>
DefDescriptor	<p>A DefDescriptor acts as a key for a definition in a registry. It's an Aura class that contains the metadata for any definition used in Aura, such as a component, action, or event. In the example of a model, it is a nicely parsed description of <code>model="java://myPackage.MyClass"</code> with methods to retrieve the language, class name, and package name. Rather than passing a more heavyweight definition around in code, Aura usually passes around a DefDescriptor instead.</p> <p>The qualified name for a DefDescriptor has a format of either <code>prefix://namespace:name</code> or <code>prefix://namespace.name</code>. For example, <code>js://ui.button</code>.</p> <ul style="list-style-type: none"> <li>• <code>prefix</code>: Defines the language, such as JavaScript or Java</li> <li>• <code>namespace</code>: Corresponds to the package name or XML namespace</li> <li>• <code>name</code>: Corresponds to the class name or local name</li> </ul>
Instance	An instance represents the data for a component, event, or action. The component data is contained in its model and attributes.
Registry	Registries store metadata definitions. Some registries last for the duration of a request, while others are cached for the lifetime of the app server. They may be created during the request process and destroyed when the server completes the request. A master definition registry contains a list of registries for each Aura resource.

## Reading Initial Component Data with Models

A model is a component's main source for dynamic data.

Use a model to read your initial component data in Aura and display the data on the user interface. You can create a model using Java or JSON. For example, a Java model could read the component's data from a database. A JSON model reads your initial component data from a JSON resource.

### Java Models

Use a Java model to read a component's data from a dynamic source, such as a database. The component generates an appropriate user interface from the model's data.

The value provider for a model is denoted by `m`. For example, the label in this button component is retrieved from the model of the component containing the `<ui:button>` tag. The value for the label is evaluated when the component renders.

```
<ui:button label="{!m.myLabel}"/>
```

On the server side, Aura's model is more of a model initializer compared to the usage of models in other MVC frameworks. The model is instantiated when the component is first requested. Perform any necessary operations to gather state, such as making database queries or external API callouts, in the model's constructor.

When the component is serialized to the client, the `@AuraEnabled` getters are executed, and their results are serialized as name-value pairs. This serialized map becomes the basis for the initial state of the model on the client.



**Note:** You can't create a new component dynamically in a model class using `Aura.getInstanceService().getInstance()`.

## Wiring Up the Model

The `aura:component` tag contains a `model` system attribute that wires it to the Java model. For example:

```
<aura:component model="java://org.auraframework.demo.notes.models.TrivialModel">
```

## Accessing the Model in Markup

Let's look at simple usage of a model in the markup of a component.

```
<aura:component model="java://org.auraframework.demo.notes.models.TrivialModel">
  <aura:attribute name="name" type="String" required="true" default="Michelle" />

  <!-- Use the "m." prefix to access any fields that are annotated with
  @AuraEnabled in the model class -->
  <h1>Title : {!m.title}</h1>

  <!-- Use v.name to directly access the component's name attribute.
  Remember that you use v to access the component's attribute values -->
  <h2>Name : {!v.name}</h2>
</aura:component>
```

The `{!m.title}` expression returns the result of the `getTitle()` getter method in the component's model class. The `getTitle()` method must be prefixed with the `@AuraEnabled` annotation.

## Java Model class

This model is simple as it doesn't read in data from a persistent data store but it demonstrates some basics, including accessing a component's attribute in the model.

```
package org.auraframework.demo.models;

import org.auraframework.instance.BaseComponent;
import org.auraframework.system.Annotations.AuraEnabled;
import org.auraframework.system.Annotations.Model;
import org.auraframework.throwable.quickfix.QuickFixException;

@Model
public class TrivialModel throws QuickFixException {

    private String title;

    // The constructor is called during the construction of each instance of the model
    // The constructor must be public
    public TrivialModel() {
        // This retrieves the component for this model as a Java object
        BaseComponent cmp =
            Aura.getContextService().getCurrentContext().getCurrentComponent();

        // Retrieve the name attribute of the component
```

```

String name = (String)cmp.getAttributes().getValue("name");

/* Do any queries or data generation in the constructor of your model.
 * In this sample, we have a trivial initialization for the title field.
 * A real-world scenario would read the data from a persistent data store. */
title = "Welcome to " + name;
}

// Use @AuraEnabled to enable client- and server-side access to the title field
@AuraEnabled
public String getTitle() {
    return title;
}
}

```

### Java Annotations

These annotations are available in Java models.

Annotation	Description
@Model	Denotes that a Java class is a model.
@AuraEnabled	Enables client- and server-side access to a getter method. This means that you only expose data that you have explicitly annotated and avoids accidentally exposing fields. Other fields are not available.

### Learn More

For a more in-depth example of a model that initializes its data from a database, see the `NoteListModel` class and the `noteList.cmp` component in the Aura Note sample app.

### See Also:

[JSON Models](#)  
[Accessing Models in JavaScript](#)  
[Creating Server-Side Logic with Controllers](#)  
[Server-Side Runtime Binding of Components](#)  
[Mocking Java Models](#)

## JSON Models

Use a JSON model to read your initial component data in Aura from a JSON resource.

To initialize your component from a more dynamic source, such as a database, use a Java model instead.

### Wiring Up the Model

There are a few ways to wire up a JSON model. A JSON model is auto-wired if it's in the component bundle and follows the naming convention, `<componentName>Model.js`.

You can explicitly declare a model in the `aura:component` tag by including a `model` system attribute with the format `model="js://<namespace>.<componentName>"`. This enables reuse of a model from another component. For example, this component uses the auto-wired model for `auradocs.sampleComponent` in `auradocs/sampleComponent/sampleComponentModel.js`.

```
<aura:component model="js://auradocs.sampleComponent
```

If you explicitly declare a `model` system attribute, it takes precedence over a model in the component bundle.



**Note:** A component can only have a JSON or Java model, but not both.

### Sample JSON Model

Here is a sample JSON model.

```
{
  "bool" : true,
  "num"  : 5,
  "str"  : "My name is JSON",
  "list" : []
}
```



**Note:** Don't use `null` for model values. Use `[]` for an empty array, `""` for an empty string, or zero for a number. This enables the framework to determine which type of value wrapper to initialize. Due to a current limitation, don't use `{ }` for an empty object.

### Accessing the Model in Markup

Here is simple usage of a model in the markup of a component.

```
<-- This component uses an auto-wired model
    as this aura:component tag has no model system attribute -->
<aura:component>
  boolean: {!m.bool}
  number:  {!m.num}
  string:  {!m.str}
  list length: {!m.list.length}
</aura:component>
```

### See Also:

[Java Models](#)

[Accessing Models in JavaScript](#)

[Component Bundles](#)

## Accessing Models in JavaScript

Use the value provider, `m`, to access a Java or JSON model in JavaScript code. For example:

```
var title = cmp.get("m.title");
alert("Title: " + title);
```

To update the model in JavaScript code, use `set ()`. For example:

```
cmp.set("m.myLabel", "updated label");
```

### See Also:

[Java Models](#)

[JSON Models](#)

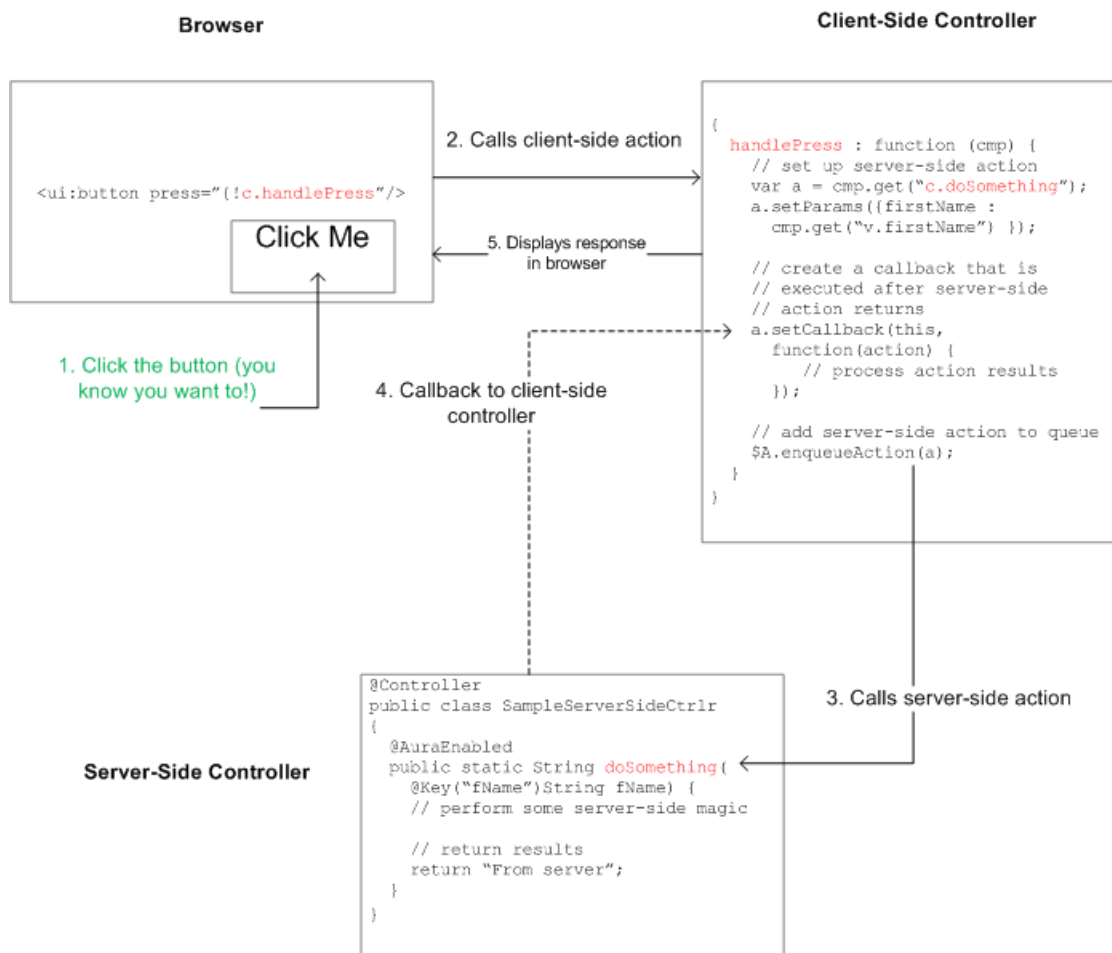
[Working with Attribute Values in JavaScript](#)

## Creating Server-Side Logic with Controllers

You can use client-side and server-side controllers in Aura. An event is always wired to a client-side controller action, which can in turn call a server-side controller action. For example, a client-side controller might handle an event and call a server-side controller action to persist data to a database.

Server-side actions need to make a round trip, from the client to the server and back again, so they are usually completed more slowly than client-side actions.

This diagram shows the flow from browser to client-side controller to server-side controller.



The `press` attribute wires the button to the `handlePress` action of the client-side controller by using `c.handlePress`. The client-side action name must match everything after the `c.`



For more details on the process of calling a server-side action, see [Calling a Server-Side Action](#) on page 131.

For an in-depth example of a server-side controller that interacts with a database, see the `NoteViewController` class in the Aura Note sample app.

### Creating a Java Server-Side Controller

Create a server-side controller in Java. Use the `@Controller` annotation before a Java class definition to denote a server-side controller.

### 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.

### 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 in Aura but it enables the framework to minimize network traffic.

### Abortable Actions

You can mark an action as abortable to make it potentially abortable while it's queued to be sent to the server or not yet returned from the server. This is useful for actions that you'd like to abort when there is a newer abortable action in the queue.

### Background Actions

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.

### Caboose Actions

Use a caboose server action to send data to the server that is not time-sensitive, such as logging, performance statistics, or click tracking data.

### Storable Actions

A server-side controller action can have its response stored in the client-side cache by the Aura framework. This can be useful if you want your app to be functional for devices that temporarily don't have a network connection.

## Creating a Java Server-Side Controller

Create a server-side controller in Java. Use the `@Controller` annotation before a Java class definition to denote a server-side controller.

All methods on server-side controllers must be static because Aura doesn't create a controller instance per component instance. Instead, all instances of a given component share one static controller.



**Warning:** Any state stored on the controller is shared across all instances of a component definition. This is unlikely to be what you want. In contrast, one model instance is created for each component instance. This means that models are the appropriate place to store state that is specific to one instance of a component.

This Java controller contains a `serverEcho` action that simply prepends a string to the value passed in. This is a simple example that allows us to verify in the client that the value was returned by the server.

```
package org.auraframework.demo.controllers;

@Controller
public class TrivialServerSideController {
```

```
//Use @AuraEnabled to enable client- and server-side access to the method
@AuraEnabled
public static String serverEcho(@Key("firstName")String firstName) {
    return ("From server: " + firstName);
}
}
```

## Java Annotations

These Java annotations are available in server-side controllers.

### @Controller

Denotes that a Java class is a server-side controller.

### @AuraEnabled

Enables client- and server-side access to a controller method. This means that you only expose data that you have explicitly annotated. Other methods are not available.

### @Key

Sets a key for each argument in a method for a server-side action. When you use `setParams` to set parameters in the client-side controller, match the JSON element name with the identifier for the `@Key` annotation. Note that we used `a.setParams({ firstName : component.get("v.firstName") })`; in the client-side controller that calls our sample server-side controller.

The `@Key` annotation means that you don't have to create an overloaded version of the method if you want to call it with different numbers of arguments. Aura simply passes in `null` for any unspecified arguments.

You can also indicate which parameters are loggable by setting the optional second attribute, `loggable`, to `true`. This example shows how to specify that the `config` and `pageSize` parameters should be included in the log:

```
public static Map<String, Object> refreshFeed(
    @Key(value = "config", loggable = true) Object config,
    @Key(value = "pageSize", loggable = true) Integer pageSize)
    throws SQLException {
    ...
}
```

### @BackgroundAction

Marks the action as a background action.

## Wiring Up a Java Server-Side Controller

The component must include a controller attribute that wires it to the server-side Java controller. For example:

```
<aura:component
    controller="java://org.auraframework.demo.controllers.TrivialServerSideController">
```

## See Also:

[Background Actions](#)

[aura:component](#)

## 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 JSON object containing name-value pairs. Each name corresponds to a client-side action. Its value is the JavaScript function associated with the action.

The following client-side controller includes an `echo` action that executes a `serverEcho` action on a server-side controller. The client-side controller sets a callback action that is invoked after the server-side action returns. In this case, the callback function alerts the user with the value returned from the server.

```
{
  "echo" : function(component) {
    // create a one-time use instance of the serverEcho action
    // in the server-side controller
    var a = component.get("c.serverEcho");
    a.setParams({ firstName : component.get("v.firstName") });

    // Create a callback that is executed after the server-side action returns
    a.setCallback(this, function(action) {
      if (action.getState() === "SUCCESS") {
        // Alert the user with the value returned from the server
        alert("From server: " + action.getReturnValue());

        // You would typically fire a event here to trigger client-side
        // notification that the server-side action is complete
      }
      else if (action.getState() === "ERROR"){
        var errors = a.getError();
        if (errors) {
          $A.logf("Errors", errors);
          if (errors[0] && errors[0].message) {
            $A.error("Error message: " + errors[0].message);
          }
        } else {
          $A.error("Unknown error");
        }
      }
      else {
        alert("Action state: " + action.getState());
      }
    });

    // 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.
    // Rather than send a separate request for each individual action,
    // Aura processes the event chain and
    // executes the action in the queue after batching up related requests.
    $A.enqueueAction(a);
  }
}
```

In the client-side controller, we use the value provider of `c` to invoke a server-side controller action. This is the same syntax as we use in markup to invoke a client-side controller action. The `cmp.get("c.serverEcho")` call indicates that we are 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.

Use `$A.enqueueAction(action)` to add the server-side controller action to the queue of actions to be executed. All actions that are enqueued this way will be run at the end of the event loop. The actions are asynchronous and have callbacks. The `runAfter` method is deprecated.

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.

**ABORTED**

The action was aborted

**See Also:**

[Handling Events with Client-Side Controllers](#)

[Queueing of Server-Side Actions](#)

## 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 in Aura but it enables the framework to minimize network traffic.

Event processing can generate a tree of events if an event handler fires more events. The framework processes the event tree and adds every action that needs to be executed on the server to a queue.

When the tree of events and all the client-side actions are processed, the framework batches actions from the queue into a message before sending it to the server. A message is essentially a wrapper around a list of actions.

There are some properties that you can set on an action to influence how Aura manages the action while it's in the queue waiting to be sent to the server. For more information, see:

- [Abortable Actions](#) on page 132
- [Background Actions](#) on page 133
- [Caboose Actions](#) on page 134
- [Storable Actions](#) on page 134

## Abortable Actions

You can mark an action as abortable to make it potentially abortable while it's queued to be sent to the server or not yet returned from the server. This is useful for actions that you'd like to abort when there is a newer abortable action in the queue.

A set of actions for a single transaction, such as a click callback, are queued together to be sent to the server. If a user starts another transaction, for example by clicking another button, all abortable actions are removed from the queue. The aborted actions are not sent to the server and their state is set to **ABORTED**. If some actions have not yet returned from the server, they

will complete, but their callbacks will not be called. An abortable action is sent to the server and executed normally unless it hasn't returned from the server when a subsequent abortable action is added to the queue.



**Note:** There is no requirement that the most recent abortable action has to be identical to the previous abortable actions. The most recent action just has to be marked as abortable.

Mark a server-side action as abortable by using the `setAbortable()` method on the `Action` object in JavaScript. For example:

```
var a = component.get("c.serverEcho");
a.setAbortable();
```

You can check for aborted actions in your callback and take appropriate action, such as logging the aborted action, if desired. For example:

```
a.setCallback(this, function(action) {
    if (action.getState() === "SUCCESS") {
        // Alert the user with the value returned from the server
        alert("From server: " + action.getReturnValue());
    }
    else if (action.getState() === "ABORTED") {
        alert("The action was aborted");
    }
    else { // something bad happened
        alert("Action state: " + action.getState());
    }
});
```

## See Also:

[Creating Server-Side Logic with Controllers](#)

[Queueing of Server-Side Actions](#)

[Calling a Server-Side Action](#)

## Background Actions

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.

Aura supports background actions as well as foreground actions, which are the default. Each background action is sent in its own request and is executed in the order that it's received. This is different from foreground actions. Multiple queued foreground actions are batched in a single request to minimize network traffic.

When the server-side actions in the queue are executed, the foreground actions are executed first and then the background actions are executed. Background actions run in parallel with foreground actions.

Aura throttles foreground and background actions separately. This means that the number of long-running background server-side actions running at a time can be controlled. Throttling is done automatically, it is not user controlled. Even with separate throttling, background actions might affect performance in some conditions, such as if the browser is doing many fetches from servers.

To set an action as a background action, get an instance of that action object in JavaScript and call the `setBackground()` method.

When `isBackground` is `true` for an action, the action can't be set back to a foreground action. In other words, calling `setBackground` to set it to `false` will have no effect.

To mark a server-side action as a background action in Java, use the `@BackgroundAction` annotation at the method level on the controller. When an action is instantiated from an `ActionDef`, the action's `isBackground` property is set to `true` automatically.

### See Also:

[Queueing of Server-Side Actions](#)

[Calling a Server-Side Action](#)

[Creating a Java Server-Side Controller](#)

## Caboose Actions

Use a caboose server action to send data to the server that is not time-sensitive, such as logging, performance statistics, or click tracking data.

A caboose action will wait until another non-caboose foreground action is sent and will piggyback on that `XMLHttpRequest` (XHR). This can improve performance by eliminating the overhead of additional round trips to the server.

When you start generating the data on the client that you want to eventually send back to the server, mark a foreground action as a caboose action with `action.setCaboose()`, enqueue the action, and set a callback with `setAllAboardCallback()`. That callback is called just before the action is sent to the server and should be written to take the data from the queue, put it in the action with one or more calls to `setParam()`, and then clear the queue. The server-side action should then process the data that was sent as parameters.



### Note:

If there is a caboose action in the queue when the user closes the app, that caboose action will not be sent.

### See Also:

[Queueing of Server-Side Actions](#)

[Calling a Server-Side Action](#)

## Storable Actions

A server-side controller action can have its response stored in the client-side cache by the Aura framework. This can be useful if you want your app to be functional for devices that temporarily don't have a network connection.



**Warning:** A storable action might result in no call to the server. An action that updates or deletes data should **never** be marked storable.

Successful actions, for which `getState()` in the JavaScript callback returns `SUCCESS`, are stored.

If a storable action is aborted after it's been sent but not yet returned from the server, its return value is still added to storage but the action callback is not called.

The action response of a storable action is saved in an internal Aura-provided storage named `actions`. This stored response is returned on subsequent calls to the same server-side action instead of the response from the server-side controller, as long as the stored response hasn't expired.

If the stored response has reached its expiration time, a new response is retrieved from the server-side controller and is stored in the `actions` storage for subsequent calls.

## Marking Storable Actions

To mark a server-side action as storable, call `setStorable()` on the action in JavaScript code, as follows.

```
a.setStorable();
```



**Note:** Storable actions are always implicitly marked as abortable too.

The `setStorable` function takes an optional parameter, which is a configuration map of key/value pairs representing the storage options and values to set. You can set only the following properties:

- `ignoreExisting`: If set to `true`, the stored item should be refreshed with a newly retrieved value, regardless of whether the item has expired or not. The default value is `false`.
- `refresh`: Overrides the item's autorefresh interval.

To set the storage options for the action response, pass this configuration map into `setStorable`.

## Refreshing an Action Response for Every Request

If a storable action returns dynamic content from the server, set the refresh interval to 0 to ensure that the data is refreshed from the server. If an action response is already cached, the cached response is displayed while the server roundtrip is happening.

To refresh the action response for each request, set:

```
a.setStorable({
    "refresh": 0
});
```

## Examples

This example marks an action as storable, forces a refresh next time the action is called, and overrides the autorefresh interval to 10 seconds.

```
a.setStorable({
    "ignoreExisting": true,
    "refresh": 10
});
```

This next example shows how to use `setStorable()` to store the server-side action response in a client-side cache. The markup includes a button that triggers the `runActionAtServerAndStore` client-side controller action. This client-side action calls a `fetchDataRecord` server-side action. Next, the action is marked as storable and is run. The server-side action return value is obtained in the callback.

This is the component markup that initializes the actions storage and contains a button.

```
<aura:component render="client" extensible="true"
    controller="java://org.auraframework.impl.java.controller.AuraStorageTestController"
    implements="auraStorage:refreshObserver">

    <auraStorage:init debugLoggingEnabled="true"
        name="actions"
        secure="true"
        persistent="false"
        clearStorageOnInit="true"
        defaultExpiration="50"
        defaultAutoRefreshInterval="60" />

    <ui:button label="Run action at Server and mark as storable"
```

```

        press="{!c.runActionAtServerAndStore}"
        aura:id="ForceActionAtServer"/>

</aura:component>

```

This is the action in the component's JavaScript client-side controller.

```

runActionAtServerAndStore: function(cmp, evt, helper){
    // Get server-side action
    var a = cmp.get("c.fetchDataRecord");

    // Set server-side action as storable
    a.setStorable();

    a.setCallback(cmp, function(a){
        var returnValue = a.getReturnValue();
    });

    // Run server-side action
    $A.enqueueAction(a);
},

```

You can also check whether an action response originates from storage by calling `isFromStorage` on the action object in the callback function of the JavaScript controller.

For a detailed description of the JavaScript API for `AuraStorageService` and `AuraStorage`, refer to the JavaScript API documentation.

### See Also:

[Calling a Server-Side Action](#)

[Caching with Storage Service](#)

[Creating Server-Side Logic with Controllers](#)

[Abortable Actions](#)

## Server-Side Rendering to the DOM

The Aura rendering service takes in-memory component state and updates the component in the Document Object Model (DOM).

The DOM is the language-independent model for representing and interacting with objects in HTML and XML documents. Aura 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:** The preferred way to customize component rendering is to use a client-side renderer. You can also use a server-side renderer but it's not recommended as they don't degrade gracefully if an error, such as a network connection outage, occurs. The framework uses a server-side renderer to render an app's template and that is the primary use case for rendering on the server.

### Creating a Java Server-Side Renderer

If you've exhausted the alternatives, including a client-side renderer, create a server-side renderer in Java by implementing the `org.auraframework.def.Renderer` interface. The interface contains one method:

```

public void render(BaseComponent<?,?> component, Appendable appendable)
    throws IOException, QuickFixException;

```



The `component` argument is the instance to render. The `appendable` argument is the output buffer.

The class that implements the interface must have a no-argument constructor. The class is instantiated as a singleton, so no state should be stored in it.

## Wiring Up a Server-Side Renderer

To wire up a server-side renderer for a component, add a `renderer` system attribute in `<aura:component>`. For example:

```
<aura:component
  renderer="java://org.auraframework.demo.notes.renderers.ReallyNeedAServerSideRenderer">
  ...
</aura:component>
```

The framework behavior is undefined if you add a server-side renderer that also includes a client-side renderer. We recommend that you use one or the other.

### See Also:

[Client-Side Rendering to the DOM](#)

[Creating App Templates](#)

## Server-Side Runtime Binding of Components

A provider enables you to use an abstract component or an interface in markup. The framework uses the provider to determine the concrete component to use at runtime.

Server-side providers are more common, but if you don't need to access the server when you're creating a component, you can use a client-side provider instead.

Set the `provider` system attribute in the `<aura:component>` tag of an abstract component or interface to point to the server-side provider Java class.

The syntax of the `provider` system attribute is `provider="java://package.class"` where `package.class` is the fully qualified name for the class.

A Java provider must:

- Include the `@Provider` annotation above the class definition
- Implement either the `ComponentDescriptorProvider` or `ComponentConfigProvider` interface



**Note:** Existing providers haven't been refactored to use the interfaces. The older method of creating a provider doesn't use interfaces and uses a static `provide()` method to return the concrete component. For an example, see `InputOptionProvider.java`. Use the provider interfaces when you are creating a new provider.

At runtime, a provider has access to a shell of the abstract component or interface, including any attribute values that have been set. The model isn't constructed yet so you can't access it. The `provide()` method can examine the attribute values that are set on the component, and return a descriptor of the non-abstract component type that should be used.



**Note:** A provider should only return concrete components that are sub-components of a single base component or implement an interface. Aura doesn't currently enforce this restriction, but will in a future release.

## ComponentDescriptorProvider

Use the `ComponentDescriptorProvider` interface to return a `DefDescriptor` describing the concrete component to use when you don't need to set attributes for the component. For example:

```
@Provider
public class SampleDescProvider implements ComponentDescriptorProvider {

    public DefDescriptor<ComponentDef> provide() {
        DefDescriptor defDesc = null;

        // logic to determine DefDescriptor to set and return.

        return defDesc;
    }
}
```

## ComponentConfigProvider

Use the `ComponentConfigProvider` interface to return a `ComponentConfig`, which describes the concrete component to use in a `DefDescriptor` and enables you to set attributes for the component. For example:

```
@Provider
public class SampleConfigProvider implements ComponentConfigProvider {

    public ComponentConfig<ComponentDef> provide() {
        ComponentConfig cmpConfig = null;

        // logic to determine DefDescriptor
        // and attributes to set.

        return cmpConfig;
    }
}
```

## Declaring Provider Dependencies

The Aura framework automatically tracks dependencies between definitions, such as components. However, if a component uses a provider that instantiates components that are not directly referenced elsewhere, use `<aura:dependency>` in the component to explicitly tell the framework about the dependency, which wouldn't otherwise be discovered by Aura.

### See Also:

[Client-Side Runtime Binding of Components](#)  
[Abstract Components](#)  
[Interfaces](#)  
[Getting a Java Reference to a Definition](#)  
[aura:dependency](#)  
[Mocking Java Providers](#)

## Serializing Exceptions

You can serialize server-side exceptions and attach an event to be passed back to the client in such a way that an event is automatically fired on the client side and handled by the client's error-handling event handler.

To do this, on the server, instantiate a `GenericEventException` that contains an event and parameters and then throw it. The exception gets serialized and when the action goes back to the client, the exception is sent along with the action as an error on the action. The status of the action will be set as "Error". The specified event in `GenericEventException` will be

fired and its handlers invoked. If a callback is provided specifically for the error state, then that callback is invoked. Otherwise, the default callback is invoked.

```
@AuraEnabled
public static void throwsGEE(@Key("event") String event, @Key("paramName") String paramName,
    @Key("paramValue") String paramValue) throws Throwable {
    GenericEventException gee = new GenericEventException(event);
    if (paramName != null) {
        gee.addParam(paramName, paramValue);
    }
    throw gee;
}
```

On the client, the client-side framework automatically handles deserializing the event and firing it. For a component event, only handlers associated with this component are invoked, else the firing of the event has no effect. For an application event, its global and all event handlers are invoked.

A `GenericEventException` is a server-side Java exception that extends the generic exception, `ClientSideEventException`. Optionally, you can extend `ClientSideEventException` yourself but it is easier to use the provided `GenericEventException`. Other classes that extend `ClientSideEventException` are the `ClientOutOfSyncException` class, the `SystemErrorException` class, the `InvalidSessionException` class, and the `NoAccessException` class. These classes are for internal use only.

For a working example of a server-side controller that throws a `GenericEventException`, refer to the `test:testActionEvent` component.

## See Also:

[Creating Server-Side Logic with Controllers](#)

# Chapter 15

## Java Cookbook

---

### In this chapter ...

- [Dynamically Creating Components in Java](#)
- [Setting a Component ID](#)
- [Getting a Java Reference to a Definition](#)

This section includes code snippets and samples that can be used in JavaScript classes.

## Dynamically Creating Components in Java

You can create a component dynamically in your Java code.

This example demonstrates how to use Java to get an instance of a component. An instance represents the data for a component. Use the `InstanceService` class to create a new component instance.

```
// listAttributes is a map of attributes for the component
Map<String, Object> listAttributes = new HashMap();
listAttributes.put("sort", "asc");
Component cmpInstance =
    Aura.getInstanceService().getInstance("auranote:noteList",
        ComponentDef.class, listAttributes);
```

The first parameter to the `getInstance` method is `auranote:noteList`, which is the qualified name for a `noteList` component in the `auranote` namespace.

The second parameter is `ComponentDef.class`, which indicates the class for the instance.

The third parameter is `listAttributes`, which contains a map of attributes for the component instance. In this case, we only have one `sort` attribute, but you can add more attributes to the map, if needed.

The `InstanceService` class also has other overloaded `getInstance` methods that take either a `Definition` or a `DefDescriptor` as their first parameter instead of a qualified name.

### See Also:

[Setting a Component ID](#)

[Component Request Glossary](#)

[Getting a Java Reference to a Definition](#)

## Setting a Component ID

To create a component with a local ID and attributes in Java code, use `ComponentDefRefBuilder` to set the component definition reference.

`ComponentDefRefBuilder` is also known as `ComponentDefRef`. The `ComponentDefRef` creates the definition of the component instance and turns it into an instance of the component during runtime.

```
ComponentDefRefBuilder builder = Aura.getBuilderService().getComponentDefRefBuilder();

//Set the descriptor for your new component
builder.setDescriptor("namespace:newCmp");

//Set the local Id for your new component
builder.setLocalId("newId");

//Set attributes on the new component
builder.setAttribute("attr1", false);
builder.setAttribute("attr2", attrVal);

//Create a new instance of the component
Component aNewCmp = builder.build().newInstance(null).get(0);
```

You can also create an instance of a component using `Aura.getInstanceService().getInstance()`, but you should use the `ComponentDefRefBuilder` if you want to:

- Set an ID on the new component.
- Set a facet on a top-level component.
- Create multiple instances of the components with minimal updates to the definition.



**Note:** The XML Parser in Aura reads in files, such as `.cmp`, `.intf`, and `.evt`, by using the `BuilderService` to construct definitions. The `BuilderService` doesn't know anything about XML. If you want to create reusable definitions that are the equivalent of what you could type into an XML file, but don't want to use XML as the storage format, use the `BuilderService`.

### See Also:

[Component Facets](#)

[Dynamically Creating Components in Java](#)

[Component Request Glossary](#)

[Server-Side Processing for Component Requests](#)

## Getting a Java Reference to a Definition

A definition in Aura describes metadata for an object, such as a component, event, controller, or model. Rather than passing a more heavyweight definition around in code, Aura usually passes around a reference, called a `DefDescriptor`, instead.

In the example of a model, a `DefDescriptor` is a nicely parsed description of `model="java://myPackage.MyClass"` with methods to retrieve the language, class name, and package name.

To create a `DefDescriptor` in Java code, use the `DefinitionService` class to create a new `DefDescriptor`.

```
DefDescriptor<ComponentDef> defDesc =  
    Aura.getDefinitionService().getDefDescriptor("ui:button", ComponentDef.class);
```

The first parameter to the `getDefDescriptor` method is `ui:button`, which is the qualified name for a button component in the `ui` namespace. The second parameter is `ComponentDef.class`, which indicates the class for the definition.

### See Also:

[Component Request Glossary](#)

# Chapter 16

## URL-Centric Navigation

---

### In this chapter ...

- [Using Custom Events in URL-Centric Navigation](#)
- [Accessing Tokenized Event Attributes](#)
- [Using Layouts for Metadata-Driven Navigation](#)

It's useful to understand how Aura handles page requests. The initial GET request for an app retrieves a template containing all the Aura JavaScript and a skeletal HTML response. All subsequent changes to everything after the # in the URL trigger an XMLHttpRequest (XHR) request for the content. The client service makes the request, and returns the result to the browser.

The portion of the URL before the # value doesn't change after the initial app request. The app is long-lived with subsequent actions causing incremental changes to the DOM for the lifetime of the app.

### Navigation Events

Aura uses its event model to manage content change in response to URL changes. The client-side `AuraHistoryService` monitors the location of the current window for changes. If the # value in a URL changes, the `AuraHistoryService` fires an Aura application event of type `aura:locationChange`. The `locationChange` event has a single attribute called `token`.

For example, if the URL changes from `/demo/test.app#` to `/demo/test.app#foo`, a `aura:locationChange` event is fired, and the `token` attribute on that event is set to `foo`.

### See Also:

[Modes Reference](#)

[aura:application](#)

[Using Layouts for Metadata-Driven Navigation](#)

[Initial Application Request](#)

## Using Custom Events in URL-Centric Navigation

If your application requires a more complex URL schema, with name-value pairs that you want to tokenize, you can extend `aura:locationChange` to add your own event type. For example, you could create the `demo/myLocationChange/myLocationChange.evt` event so that Aura automatically parses the `thing1` and `thing2` attributes in the URL.

```
<aura:event type="application" extends="aura:locationChange">
  <aura:attribute name="thing1" type="String"/>
  <aura:attribute name="thing2" type="Boolean"/>
</aura:event>
```

Update the `locationChangeEvent` attribute in your `<aura:application>` component to indicate to `AuraHistoryService` that you want to parse the hash of the URL into the custom event.

```
<aura:application locationChangeEvent="demo:myLocationChange">
```

Now, when the URL changes to `/demo/test.app#foo?thing1=Howdy&thing2=true`, the `AuraHistoryService` fires an event of type `demo:myLocationChange` with token set to `foo`, `thing1` set to `Howdy` and `thing2` set to `true`.



**Note:** The attributes after the `#` value use the same format as a query string: `#foo?thing1=Howdy&thing2=true`.

However, a real request query string starts before the `#` value. A sample query string that sets the Aura mode to `PROD` (production) is `/demo/test.app?aura.mode=PROD&queryStrParam2=val2#foo`.

## Accessing Tokenized Event Attributes

To see how you'd access the tokenized attributes, imagine a scenario where a component uses a `getHomeComponents` server-side action to retrieve components. You can write the `getHomeComponents` action to accept arguments that match the attributes in your custom location change event. The arguments are automatically mapped from the location change event to the action call.

```
@AuraEnabled
public static Aura.Component[] getHomeComponents(String token, String thing1, Boolean
thing2){...}
```

## Using Layouts for Metadata-Driven Navigation

Layouts are a metadata-driven description of navigation in an application. You can describe in an XML file how you want the application to respond to changes to everything after the `#` (hash) in the URL. You can use Aura without layouts, but they offer a centralized location for managing URL-centric navigation.

### Layouts Metadata File

Each app can have a layouts file that describes navigation in the app. The name of the layouts file is derived from the name of the app. If the app is `demo.app`, the layouts file is `demoLayouts.xml` and it's in the same directory as `demo.app`.

The layouts file contains a `<aura:layouts>` system tag that can contain one or more `<aura:layout>` system tags. Each `<aura:layout>` is a matching rule for everything after the `#` in the URL.



A `<aura:layout>` system tag contains one or more `<aura:layoutItem>` system tags. Each `<aura:layoutItem>` is a template that populates a container with markup or the results of an action.

A container is dynamically populated when a `<aura:layout>` tag is matched. The container must have a `aura:id` attribute that is findable in the app. If an app contains components with attributes of `aura:id="sidebar"` and `aura:id="content"`, you can refer to the sidebar and content containers in a `<aura:layoutItem>` tag in the layouts file.

For example, consider `demo.app` that contains components with attributes of `aura:id="sidebar"` and `aura:id="content"`.

```
<aura:set attribute="left">
  <div class="sidebar" aura:id="sidebar"></div>
</aura:set>
<div class="content" aura:id="content"></div>
```

You can refer to the sidebar and content containers in a `<aura:layoutItem>` system tag in the associated `demoLayouts.xml` file.

```
<aura:layout name="sample" match="^{.}{3}$">
  <aura:layoutItem container="sidebar" action="{!c.getList}"/>
  <aura:layoutItem container="content"><p>You can put markup here though it normally
would be dynamic</p></aura:layoutItem>
</aura:layout>
```

Each `<aura:layout>` first compares everything after the # in the URL with the name attribute. If it doesn't match, it compares the # value in the URL with the optional match attribute, which is a regular expression. In this case, the name attribute would match `#sample` in the URL and would then fall back to matching any three characters based on the `^{.}{3}$` regular expression.

You would normally have more than one `<aura:layout>` in a layouts file. In that case, Aura attempts to match against each name attribute first. If there is no match for any `<aura:layout>`, the framework attempts to match against each match attribute.

The first `<aura:layoutItem>` populates the sidebar container with the results of the `getList` action on the server-side controller. The controller is defined in the `<aura:application>` tag in the associated `.app` file. A controller action is useful in this scenario because returning a list may need security checks and other processing that can't be expressed statically in markup.



**Note:** You can only reference a server-side action in a `<aura:layoutItem>`. You can't reference a client-side action.

The next `<aura:layoutItem>` populates the content container with some markup. In this case, it's static markup, but it would usually be dynamic and can include any Aura components.

The `<aura:layouts>` system tag supports the following optional attributes:

#### **default**

This is the layout to use when the app is initially loaded and there is no # value in the URL.

#### **catchall**

This is the layout to use when the # value doesn't match any `<aura:layout>`.

### **Using Custom Events in Metadata-Driven Navigation**

#### **See Also:**

[URL-Centric Navigation](#)

## Using Custom Events in Metadata-Driven Navigation

We saw how to create a custom event to tokenize multiple name-value pairs in [URL-Centric Navigation](#) on page 143. If your app has a layouts file, the layout service automatically handles the location change events that are fired.

Let's look at how layouts work with the same `demo/myLocationChange/myLocationChange.evt` event.

```
<aura:event type="application" extends="aura:locationChange">
  <aura:attribute name="thing1" type="String"/>
  <aura:attribute name="thing2" type="Boolean"/>
</aura:event>
```

To see how you'd access the tokenized attributes, let's look at a `<aura:layoutItem>` that uses a server-side action to retrieve components.

```
<aura:layoutItem container="center" action="{!c.getHomeComponents}"/>
```

You can write the `getHomeComponents` action to accept arguments that match the attributes in your custom location change event. The arguments are automatically mapped from the location change event to the action call.

```
@AuraEnabled
public static Aura.Component[] getHomeComponents(String token, String thing1, Boolean
thing2){...}
```

# Chapter 17

## Using Object-Oriented Development

---

### In this chapter ...

- [What is Inherited?](#)
- [Inheritance Rules](#)
- [Inherited Component Attributes](#)
- [Accessing a Super Component](#)
- [Abstract Components](#)
- [Interfaces](#)

Aura provides the basic constructs of inheritance, polymorphism, 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 strong separation gives component authors freedom to change the internal implementation details and insulates component consumers from those changes.

You can extend a component, app, interface or an event, or you can implement a component interface in Aura.

## What is Inherited?

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

### Component Attributes

All attributes are inherited. You can override an attribute in a sub-component using `<aura:attribute>`. However, you should only do this if you want to change the attribute's default value in the sub-component.

Use `<aura:set>` in a sub-component to set an attribute in a super component when you are extending a component or implementing an interface.

### Events

A child component that extends a super component can handle events fired by the super component. The child 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 child component. Aura doesn't guarantee the order of event handling.

When an event fires, handlers for the event are executed. Handlers for any event that extend the event are also fired.

### Helpers

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

### Controllers

A child 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 child 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.

### Models Are Not Inherited

A component's model is **not** inherited by a component that extends the super component.

#### See Also:

[Component Attributes](#)

[Events](#)

[Sharing JavaScript Code in a Component Bundle](#)

[Handling Events with Client-Side Controllers](#)

[aura:set](#)

[Java Models](#)

## Inheritance Rules

This table describes the inheritance rules for various elements in Aura.

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

**See Also:**[Interfaces](#)[Events](#)

## Inherited Component Attributes

Inherited attributes behave differently in Aura than, for example, inherited class fields in Java. In Aura, an attribute that is inherited from a base component can have different values in the sub-component and the base component. This will be clearer when we walk through an example.

Use <aura:set> in a sub-component to set the value of any attribute on the super component or to set an attribute on a component reference.

We will be looking at the `body` attribute for each of our sample components so now is a good time for a quick refresher. <aura:component> has a `body` attribute that is inherited by all components. 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">. `{!v.body}` outputs the body of the component.

The default renderer for a component iterates through its `body` attribute, renders everything, and passes the rendered data to its super component. 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.

`auradocs:parent` is the parent or super component. It inherently extends <aura:component>.

**parent.cmp**

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

At this point, `auradocs:parent` doesn't render the `body` attribute since we haven't set it yet.

You can only extend a component that has its `extensible` system attribute explicitly set to `true`. The `extensible` system attribute is defined in <aura:component>. `sampleBase.cmp` can be extended because it sets `extensible="true"`.

`auradocs:child` extends `auradocs:parent` by setting `extends="auradocs:parent"` in its <aura:component> tag.

**child.cmp**

```
<aura:component extends="auradocs:parent">
  Child body: {!v.body}
</aura:component>
```

auradocs:child renders this body value.

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

In other words, auradocs:child sets the body attribute of its super component, auradocs:parent.

auradocs:container contains a reference to auradocs:child.

**container.cmp**

```
<aura:component>
  <auradocs:child>
    Body value
  </auradocs:child>
</aura:component>
```

In auradocs:container, we set the body attribute of auradocs:child to Body value. auradocs:container renders this body value.

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

**See Also:**

[aura:set](#)

[Component Body](#)

[Server-Side Rendering to the DOM](#)

[aura:component](#)

## Accessing a Super Component

You can use the `super` value provider in markup to access a super component. Use `getSuper()` in JavaScript code. For example, if a sub component needs to access an attribute in its super component, use `cmp.getSuper().get("v.parentAttributeName")`, where `parentAttributeName` is an attribute in the super component.



**Note:** We don't recommend using `super` and `getSuper()` unless you must as it reduces component encapsulation. It's rare to have to use this syntax but it can be useful when you are writing tests.

## Traversing a Component's Extension Hierarchy

When you instantiate a component that extends a super component, the super component is instantiated as a separate component object. You can access the super component by calling `getSuper()` on the component.

For the rest of this topic, when we refer to a component, it could be a component or an app.

Some of the attributes in a component may be inherited from super components. A super component may have different values for attributes with the same name, or may also have values for attributes that we have no value for at this level of the extension hierarchy.

Now that we know that for any given component, an attribute may have different values at different extension levels, let's look at a JavaScript code sample that traverses to the root of the extension hierarchy and looks at the `body` attribute for that `<aura:component>`.

```
var cmp;
var superCmp = cmp.getSuper();
while (superCmp) {
    cmp = superCmp;
    superCmp = cmp.getSuper();
}

// Now, cmp points to the root of the hierarchy.
// We can get the body array from it.
var bodyArray = cmp.get("v.body");
for (var i=0; i < bodyArray.length; i++) {
    var bodyCmp = bodyArray[i];
    // do something with this component
}
```

#### See Also:

[Component Body](#)

## 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, Aura 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 either extend it and fill out the remaining implementation, or add a provider. An abstract component can't be used directly in markup unless you define a provider.

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

#### See Also:

[Server-Side Runtime Binding of Components](#)

[Interfaces](#)

## Interfaces

Object-oriented languages, such as Java, support the concept of an interface that defines a set of method signatures. A class that implements the interface must provide the method implementations. An interface in Java can't be instantiated directly, but a class that implements the interface can.

Similarly, Aura supports the concept of interfaces that define a component's shape by defining its attributes.

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.

An Aura interface starts with the `<aura:interface>` tag. It can only contain `<aura:attribute>` tags that define the interface's attributes. You can't use markup, renderers, controllers, models or anything else in an interface.

To use an interface, you must implement it or add a provider. An interface can't be used directly in markup otherwise. 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" >
```



**Note:** Use `<aura:set>` in a sub-component to set the value of any attribute that is inherited from the parent component. This works for components and abstract components, but it doesn't work for interfaces. To set the value of an attribute inherited from an interface, you must redefine the attribute in the sub-component using `<aura:attribute>` and set the value in its default attribute.

### See Also:

[Server-Side Runtime Binding of Components](#)

[aura:set](#)

[Abstract Components](#)

## Marker Interfaces

You can use an interface as a marker interface that is implemented by a set of components that you want to easily identify for specific usage in your app.

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

In Java, use the `isInstanceOf()` method in the `ComponentDef` or `ApplicationDef` interfaces.



# Chapter 18

## Caching with Storage Service

### In this chapter ...

- [Initializing Storage Service](#)
- [Using Storage Service](#)

Aura Storage Service provides a powerful, simple-to-use caching infrastructure for Aura clients. Aura client applications can benefit from caching data to reduce response times of pages by storing and accessing data locally rather than requesting data from the server. This enhances the user experience on the client. Caching is especially beneficial for high-performance, mostly connected applications operating over high latency connections, such as 3G networks.

The advantage of using Aura Storage Service instead of other caching infrastructures, such as Apple local storage for iOS devices, is that Aura Storage Service offers several types of storage through adapters. 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
SmartStore	true	true
WebSQL	true	false
MemoryAdapter	false	true

### SmartStore

(Persistent and secure) The SmartStore caching service is provided by the Salesforce Mobile SDK and is available only if you have installed the Salesforce Mobile SDK. The Salesforce Mobile SDK enables developing mobile applications that integrate with Salesforce. You can use SmartStore with these mobile applications for caching data.

### WebSQL

(Persistent but not secure) Provides access to a client-side SQL database.

### MemoryAdapter

(Not persistent but secure) Provides access to the JavaScript main memory space for caching data. The stored cache persists only per browser page. Browsing to a new page resets the cache. Also, MemoryAdapter provides cache management capabilities. If the memory size limit has been reached, MemoryAdapter removes the least recently used data from the cache to shrink the cache size.

Aura 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 secure storage service, Aura Storage Service will return the SmartStore storage.

There are two types of storage:

- Custom named storage: Storage that you control by adding and retrieving items to and from storage.
- Aura-provided actions storage: Storage that is available for client-side and server-side actions that enables caching action response values.

When you initialize Aura storage, you can set certain options, such as the maximum cache size and the default expiration time. The storage name is required and must be specified.



**Note:** The name of Aura storage can be any name except for “actions”, which is reserved for the server action storage that the Aura framework uses.

The expiration time for an item in storage specifies the duration after which an item should be replaced with a fresh copy. The refresh interval takes effect only if the item hasn't expired yet and applies to the actions storage only. In that case, if the refresh interval for an item has passed, the item gets refreshed after the same action is called. If stored items have reached their expiration times or have exceeded their refresh intervals, they're replaced only after a call is made to access them and if the client is online.

#### See Also:

[\*Creating Server-Side Logic with Controllers\*](#)

[\*Storable Actions\*](#)

[\*Initializing Storage Service\*](#)

## Initializing Storage Service

To use storage, you must initialize it first and specify a name and, optionally, other properties. If you don't specify the optional properties, the Aura Storage Service uses default values set by the `initStorage()` JavaScript API of [AuraStorageService](#).

You can initialize storage for your component using markup in one of two ways: by using a template or by adding the markup in the component body.

This example shows how to use a template to initialize storage using Aura markup. The component references the template in the `template` attribute. The template defined in the second example contains `auraStorage:init` tags that specify storage initialization properties. This example initializes three different storages: the Aura-provided actions storage, and two custom storages named `savings` and `checking`.

```
<aura:component render="client" template="auraStorageTest:namedStorageTemplate">
</aura:component>

<aura:component isTemplate="true" extends="aura:template">
  <aura:set attribute="auraPreInitBlock">
    <auraStorage:init name="actions" persistent="false" secure="false" maxSize="9999"/>

    <auraStorage:init name="savings" persistent="false" secure="true" maxSize="6666"/>

    <auraStorage:init name="checking" maxSize="7777"/>
  </aura:set>
</aura:component>
```

Alternatively, you can add `auraStorage:init` tags directly in the body of your component definition. The following example shows component markup that initializes a storage named `savings`.

```
<aura:component render="client" extensible="true"
  controller="java://org.auraframework.impl.java.controller.AuraStorageTestController"
  implements="auraStorage:refreshObserver">

  <auraStorage:init debugLoggingEnabled="true"
    name="savings"
    secure="true"
    persistent="false"
    clearStorageOnInit="true"
    defaultExpiration="50"
    defaultAutoRefreshInterval="60" />

</aura:component>
```

Alternatively, you can initialize storage on-the-fly using the JavaScript API. This example shows how to initialize Aura Storage Service using `initStorage()` in a JavaScript client-side controller.

```
var storage = $A.storageService.initStorage("MyStorage", // name
  true, // persistent
  true, // secure
  512, // maxSize
  600, // defaultExpiration
  600, // defaultAutoRefreshInterval
```

```

true,          // debugLoggingEnabled
true);        // clearStorageOnInit

```

**See Also:**[Storable Actions](#)[Using Storage Service](#)

## Using Storage Service

After you've initialized your custom storage, you can add and retrieve items from your storage. To do so, use the JavaScript `put` and `get` API of [AuraStorage](#).



**Note:** The Aura-provided actions storage for server-side actions automatically adds and retrieves items from storage and doesn't require you to call `put` and `get` explicitly.

This example shows how to use the storage object returned by the previous example to store items. The call to `put` takes a key that is used to uniquely identify the stored item.

```

var value1 = 67;
storage.put("score", value1);
storage.put("name", "joe smith");

```

You can retrieve stored items by using the `get` method. The parameters of the `get` method are the key of the value to retrieve and a callback function. The callback function is called asynchronously and has the item that was fetched from the storage as its parameter.

```

storage.get("score", function(item) { var myRetrievedScore = item; });
storage.get("name", function(item) { console.log(item); });

```

You can obtain any initialized named storage by calling `getStorage()` and by passing it the storage name. For example:

```

var storage = $A.storageService.getStorage("MyStorage");

```



**Note:** The `getName()` method returns the type of storage selected, not the name of the storage.

There are other methods you can call on the storage object. For a detailed description of the JavaScript API for [AuraStorageService](#), see [AuraStorageService](#) and for [AuraStorage](#), see [AuraStorage](#).

For example, you can get the current cache size and clear the storage, as follows.

```

// Get cache size
var size = $A.storageService.getStorage("MyStorage").getSize();
// Clear the cache
$A.storageService.getStorage("MyStorage").clear();

```

**See Also:**[Storable Actions](#)[Initializing Storage Service](#)

# Chapter 19

## Using the AppCache

---

### In this chapter ...

- [Enabling the AppCache](#)
- [Loading Resources with AppCache](#)
- [Specifying Additional Resources for Caching](#)

Application cache (AppCache) speeds up app response time and reduces server load by only downloading resources that have changed. It improves page loads affected by limited browser cache persistence on some devices.

AppCache can be useful if you're developing apps for mobile devices, which sometimes have very limited browser cache. Apps built for desktop clients may not benefit from the AppCache. Aura supports AppCache for WebKit-based browsers, such as Chrome and Safari.



**Note:** See [an introduction to AppCache](#) for more information.

### See Also:

[Component Request Overview](#)  
[aura:application](#)

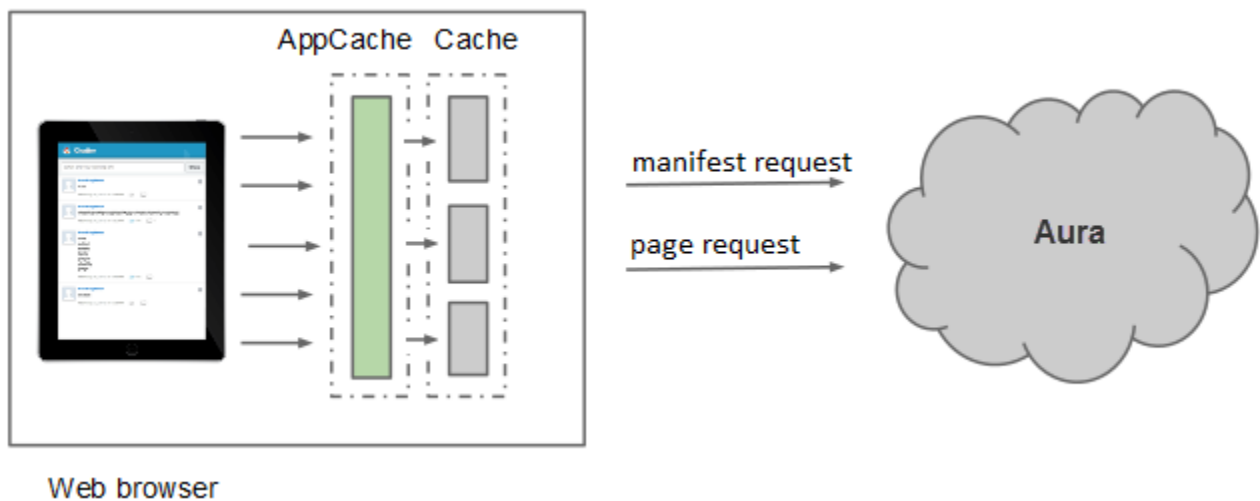
## Enabling the AppCache

Aura disables the use of AppCache by default.

To enable AppCache in your application, set the `useAppcache="true"` system attribute in the `aura:application` tag. We recommend disabling AppCache during initial development while your app's resources are still changing. Enable AppCache when you are finished developing the app and before you start using it in production to see whether AppCache improves the app's response time.

## Loading Resources with AppCache

A cache manifest file is a simple text file that defines the Web resources to be cached offline in the AppCache.



The cache manifest is auto-generated for you at runtime if you have enabled AppCache in your application. If there are any changes to the resources, Aura updates the timestamp to trigger a refetch of all resources.

When a browser initially requests an app, a link to the manifest file is included in the response. The browser retrieves the resource files that are listed in the manifest file, such as the JavaScript and CSS files, and they are cached in the browser cache. Finally, the browser fetches a copy of the manifest file and downloads all resources listed in the manifest file and stores them in the AppCache.

## Specifying Additional Resources for Caching

When AppCache is enabled, you can specify web resources to be cached in addition to the resources that Aura caches by default.

These additional resources can be any resources that can be referenced and cached, such as JavaScript (`.js`) files, CSS stylesheet (`.css`) files, and images.

To specify additional resources for the AppCache, add the `additionalAppCacheURLs` system attribute to the `aura:application` tag in your `.app` file. The `useAppcache="true"` attribute must be also set to enable caching. The `additionalAppCacheURLs` attribute value holds the URLs of the additional resources. The URLs can be local, such as `"/resources/format.css"`, or absolute, such as `"http://example.com/resources/format.css"`. When specifying more than one resource, separate the resources with commas.

This is an example of using the `additionalAppCacheURLs` attribute in the application tag. In this example, the URLs in the attribute value are obtained from a server controller action.

```
<aura:application useAppcache="true" render="client" access="global"
    controller="java://org.auraframework.impl.java.controller.TestController"
    additionalAppCacheURLs="{!c.getAppCacheUrls}">
</aura:application>
```

This is the implementation of the server controller action.

```
@AuraEnabled
public static List<String> getAppCacheUrls() throws Exception {
    List<String> urls = Lists.newArrayList();
    urls.add("/auraFW/resources/aura/auraIdeLogo.png");
    urls.add("/auraFW/resources/aura/resetCSS.css");
    return urls;
}
```

# Chapter 20

## Controlling Access

### In this chapter ...

- [Application Access Control](#)
- [Interface Access Control](#)
- [Component Access Control](#)
- [Attribute Access Control](#)
- [Event Access Control](#)

Aura enables you to control access to your applications, interfaces, components, attributes, and events via the `access` attribute on these tags.

Tag	Description
<code>aura:application</code>	Represents an Aura application
<code>aura:interface</code>	Represents an Aura interface
<code>aura:component</code>	Represents an Aura component
<code>aura:attribute</code>	Represents an Aura attribute in an application, interface, component, or event
<code>aura:event</code>	Represents an Aura event

By default, the `access` attribute is set to `internal` for all tags.



## Application Access Control

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

Possible values are listed below.

Modifier	Description
<code>global</code>	The app can be extended by another app in any namespace if <code>extensible="true"</code> is set on the <code>aura:application</code> tag.
<code>public</code>	The app can be extended by another app within the same namespace only.
<code>internal</code>	The app can be extended by another app in another system namespace. A system namespace is a privileged namespace that has access to all components. This is the default access level. If set to <code>internal</code> , the app isn't directly accessible via a URL in <code>PROD</code> mode.

## Interface Access Control

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

Possible values are listed below.

Modifier	Description
<code>global</code>	The interface can be extended by another interface or used by a component in any namespace.
<code>public</code>	The interface can be extended by another interface or used by a component within the same namespace only.
<code>internal</code>	The interface can be extended by another interface or used by a component in another system namespace. A system namespace is a privileged namespace that has access to all components. This is the default access level.

An interface can extend another interface but a component can't extend an interface. 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 indicates whether the component can be extended or used outside of the component's namespace.

Possible values are listed below.

Modifier	Description
<code>global</code>	The component can be used by another component or application in any namespace. It can also be extended in any namespace if <code>extensible="true"</code> is set on the <code>aura:component</code> tag.

Modifier	Description
<code>public</code>	The component can be extended or used by another component, or used by an application within the same namespace only.
<code>internal</code>	The component can be extended or used by another component in another system namespace. A system namespace is a privileged namespace that has access to all components. This is the default access level. If set to <code>internal</code> , the component isn't directly accessible via a URL in <code>PROD</code> mode.

## Attribute Access Control

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

Possible values are listed below.

Access	Description
<code>global</code>	The attribute can be used in any namespace.
<code>public</code>	The attribute can be used within the same namespace only.
<code>private</code>	The attribute can be used only within the container app, interface, component, or event, and can't be referenced externally.
<code>internal</code>	The attribute can be used in another system namespace. A system namespace is a privileged namespace that has access to all components. This is the default access level.

## Event Access Control

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

Possible values are listed below.

Modifier	Description
<code>global</code>	The event can be used or extended in any namespace.
<code>public</code>	The event can be used or extended within the same namespace only.
<code>internal</code>	The event can be used in another system namespace. A system namespace is a privileged namespace that has access to all components. This is the default access level.

## Chapter 21

### Testing Components

---

#### In this chapter ...

- [JavaScript Test Suite Setup](#)
- [Assertions](#)
- [Debugging Components](#)
- [Utility Functions](#)
- [Sample Test Cases](#)
- [Mocking Java Classes](#)

Aura's loosely coupled components facilitate maintainability and enable efficient testing. Components are isolated from their application context for easier testing. Aura supports JavaScript testing for components and applications in production mode.

Add component tests to a JavaScript file in the component bundle. For example, a component `myData.cmp` in the `myApp` namespace is saved in the folder `myData`, which can contain a test file `myDataTest.js`.

To reuse code among test cases, use the `setUp` and `tearDown` functions, which can be useful for quickly setting up or removing objects. They are called before and after a test method is run. During test execution, additional suite methods can be accessed with `this.sharedMethod()`.



**Note:** You can view Aura's test methods in the [JavaScript API reference](#).

Assertions and utility functions are also available for unit testing.

Run JavaScript tests in a Web browser by appending `?aura.mode=JSTEST` to your production component. For example, if you have a component `myData.cmp` in the `myApp` namespace, you can run test cases on `http://<your server>/myApp/myData.cmp?aura.mode=JSTEST`.

#### See Also:

[Component Bundles](#)

[Modes Reference](#)

[Querying State and Statistics](#)

[Assertions](#)

[Utility Functions](#)

## JavaScript Test Suite Setup

A test file in a component bundle contains a suite of tests and properties, where each function represents a different test case.

You would typically define any shared properties before your test cases. Your test functions must follow the naming convention `test<testName>`. A basic test suite looks like this.

```
{
  /** Properties shared across test cases */
  attributes: {
    label: 'Submit',
    //Other attributes here
  },
  browsers: ['GOOGLECHROME', 'SAFARI', 'IPAD' ],
  setUp: function(component){
    //Runs before each test case is executed but after component initialization
  },
  tearDown: function(component){
    //Runs after each test case is executed
  },
  sharedMethod: function(arg1, arg2){
    //Utility functions that are invoked by calling this.sharedMethod(x, y)
  },

  /** Test Cases */
  testCases: {
    attributes: {
      //Attributes
    },
    browsers: [ '-FIREFOX' ],
    test: [ //A single function or a list of functions
      function(component){
        //Test something
      },
      function(component){
        //Test something
      }
    ]
  }
}
```

The `attributes` property specifies the attribute values that the component to be tested should be instantiated with. The `attributes` and `browsers` properties are optional.

### Test Suite Properties

Test suite properties are values that the target component are instantiated with. The following lists supported properties for a test suite.

#### **attributes**

Attribute values are applied to all test cases in the suite. They are passed as query parameters in the initial GET request. For example, this code initializes the `label` and `buttonTitle` attributes on a `ui:button` component.

```
attributes:{
  label: 'Submit',
  buttonTitle: 'click once'
}
```

**browsers**

List browsers you want all test cases to test against. If this property is not specified, the tests execute in all supported browsers. Values prefixed with a hyphen exclude that browser from the test.

```
browsers: [ 'GOOGLECHROME', 'SAFARI', '-IPAD' ]
```

**setUp**

This property executes before each test case but after the component has been initialized.

**tearDown**

This property executes after each test case, regardless of the test status.

**sharedMethod**

Put additional utility functions here if your test needs to access them. This example is invoked with `this.sharedMethod(x, y)`.

```
sharedMethod: function(argument1, argument2){
  $A.test.assertNotNull(argument1, 'The first argument recieved was null');
}
```

**mocks**

Mocking isolates your JavaScript tests from other resources, such as a Java model, provider, or server-side controller. Mocks that are defined as a suite property are shared among all test cases. For more information, see [Mocking Java Classes](#) on page 170.

**Test Cases**

Test cases are typically defined after the suite properties. They contain the `attributes`, `browsers`, and `mocks` properties. If these properties are specified in a test case, their values override those provided by suite properties. Additionally, a test case can contain a `test` property that's defined with a function or a list of functions.

```
test: [
  function(component) {
    $A.test.assertTrue(true, 'This obviously should have passed. ');
    $A.test.assertEquals( 'Opt Out', component.get("v.label"), "Wrong label.");
    $A.test.assertEquals( 'click once', component.get("v.buttonTitle"), "Wrong
tooltip.");
    debugger; // Break at this point in browsers that support the directive
    component.get('e.press').fire();
    $A.test.waitFor('expected', function() {
      var lookForTextAfterClick = component.get('v.updatedOnClick');
      return lookForTextAfterClick;
    });
  }, function(component) {
    $A.test.assertTrue(true, 'This also obviously should have passed after the click. ');
  }
]
```

After the first function runs, the test waits for `$A.test.waitFor` to complete. This method compares `expected` and the return value of the `lookForTextAfterClick`. When this comparison evaluates to true, the next function is run.

## Assertions

Assertions evaluate an object or expression for expected results and are the foundation of component testing. Each JavaScript test can contain one or more assertions. The test passes only when all the assertions are successful. Assertions should be prefixed with `aura.test` or `$A.test`. If an assertion fails, an error message is typically returned with the `assertMessage` or `errorMessage` string.

Aura supports the following assertions.

Assertion	Description
<code>aura.test.assert(condition, assertMessage)</code>	Asserts that the condition is <code>true</code> .
<code>aura.test.assertAccessible(errorMessage)</code>	Asserts that the HTML output of the target component is accessibility compliant.
<code>aura.test.assertDefined(arg1, assertMessage)</code>	Asserts that <code>arg1</code> is defined.
<code>aura.test.assertEquals(arg1, arg2, assertMessage)</code>	Asserts that <code>arg1 === arg2</code> is true, where <code>arg1</code> is the expected value and <code>arg2</code> is the actual value.
<code>aura.test.fail(assertMessage)</code>	Throws an error with the <code>assertMessage</code> string. Use this to test error handling. For example: <pre> try {     // do something where you expect an     error     aura.test.fail("should have got an error"); } catch(e) {     // assert expected error } </pre>
<code>aura.test.assertFalse(condition, assertMessage)</code>	Asserts that the condition is false.
<code>aura.test.assertFalsy(condition, assertMessage)</code>	Asserts that the condition is false, null, or undefined.
<code>aura.test.assertNotEquals(arg1, arg2, assertMessage)</code>	Asserts that <code>arg1 === arg2</code> is false, where <code>arg1</code> is the expected value and <code>arg2</code> is the actual value..
<code>aura.test.assertNull(arg1, assertMessage)</code>	Asserts that <code>arg1</code> is null. If it's not null, throws an error with the <code>assertMessage</code> string.
<code>aura.test.assertStartsWith(start, full, assertMessage)</code>	Asserts that the <code>full</code> string starts with the <code>start</code> string.
<code>aura.test.assertNotNull(arg1, assertMessage)</code>	Asserts that <code>arg1</code> is not null.
<code>aura.test.assertTrue(condition, assertMessage)</code>	Asserts that the condition is <code>true</code> . This is the same as <code>aura.test.assert(condition, assertMessage)</code> .
<code>aura.test.assertTruthy(condition, assertMessage)</code>	Asserts that the condition is <code>true</code> , null, or defined.
<code>aura.test.assertUndefined(arg1, assertMessage)</code>	Asserts that the argument is undefined.

Assertion	Description
<code>aura.test.assertUndefinedOrNull (arg1, assertMessage)</code>	Asserts that the argument is undefined or null.
<code>aura.test.assertNotUndefinedOrNull (arg1, assertMessage)</code>	Asserts that the argument is not undefined or not null.

Include unique and specific error messages in your assert statements. For example, use `assertTrue (run, "Returns true if the action has run successfully.")` instead of a generic message. Making each assert message unique also helps in narrowing down which assert statement has failed.



**Note:** Refer to the JavaScript API reference for a full list of assertions.

### See Also:

[Supporting Accessibility](#)

## Debugging Components

Use the `debugger;` statement to debug your JavaScript tests. Remove or comment out the `debugger;` statement after you finish debugging.

You can view your debug output by appending `?aura.mode=JSTESTDEBUG` to your production component, which has minimal formatting for readability. Otherwise, append `?aura.mode=JSTEST` for a minified debug output.

Another useful tool for debugging is Google Chrome's Developer Tools.

- To open Developer Tools on Windows and Linux, press Control - Shift - I in your Chrome browser.
- To quickly find which line of code a test fails on, enable the **Pause on all exceptions** option before running the test.

Learn more about how to use breakpoints at [Chrome Developer Tools](#).

To simulate a user interaction, fire the associated Aura event. For example, use `buttonComponent.get("e.press").fire()` to simulate a button click event.

### See Also:

[Debugging](#)


[Events](#)

[Modes Reference](#)

## Utility Functions

Utility functions provides additional support for Aura's unit testing and should be prefixed with `aura.test` or `$A.test`.

Utility	Description
<code>aura.test.addFunctionHandler (instance, originalFunction, newFunction, postProcess)</code>	Adds a new function handler and overrides the original function.
<code>aura.test.addWaitFor (expected, testFunction, callback)</code>	Waits for <code>expected === testFunction().callback</code>

Utility	Description
<code>aura.test.callServerAction(action, doImmediate)</code>	Runs a server action. The test waits for any actions to complete before running the next function. If <code>doImmediate</code> is set to true, the request is sent immediately. Otherwise, the action is queued after prior requests.
<code>aura.test.getErrors()</code>	Returns errors as JSON encoded strings. If no errors are found, return an empty string.
<code>aura.test.getOuterHtml(node)</code>	Returns the outer HTML of an element.
<code>aura.test.getPrototype(instance)</code>	Returns the prototype of the instance or object.
<code>aura.test.getAction(component, name, params, callback)</code>	Returns an instance of an action.
<code>aura.test.getText(node)</code>	Returns text as a string.
<code>aura.test.isComplete()</code>	Returns whether the test is finished running.
<code>aura.test.overrideFunction(instance, originalFunction, newFunction)</code>	Overrides an existing function.
<code>aura.test.print(value)</code>	<p>Returns the value cast to a string.</p> <p>Possible return values are:</p> <ul style="list-style-type: none"> <li>• <code>undefined</code></li> <li>• <code>null</code></li> <li>• <code>"value"</code>—the value cast to a string</li> <li>• <code>value.toString()</code>—for non-strings</li> </ul>
<code>aura.test.runAfterIf(conditionFunction, callback, intervalInMs)</code>	<p>Evaluates <code>conditionFunction</code> every interval. When it returns a truthy value, execute the callback. <code>intervalInMs</code> is 500 milliseconds by default.</p> <div>  <p><b>Note:</b> Most values in JavaScript are truthy, such as objects, arrays, non-zero numbers, and non-empty strings.</p> </div>
<code>aura.test.select()</code>	Returns a list of elements within the document that matches the given arguments.
<code>aura.test.setTestTimeout(timeoutMsec)</code>	Sets the timeout in milliseconds from now.



**Note:** Refer to the JavaScript API reference for a full list of utility methods and arguments.

## See Also:

[Assertions](#)



## Sample Test Cases

The following test case uses the utility function `runAfterIf` and assert statements to check that the right buttons are displayed in order.

```
testButtons:{
  test:function(cmp){
    // render Cancel and Save buttons in view
    $A.test.runAfterIf(
      //check condition function before running callback function
      function(){ return buttons[0].getElement() !== null; },
      function(){
        // check the right buttons are displayed
        $A.test.assertEquals(2, buttons.length, "expected cancel and save");
        $A.test.assertEquals("Cancel", buttons[0].find("div").getElement().innerText,
          "Cancel button not first");
        $A.test.assertEquals("Save", buttons[1].find("div").getElement().innerText,
          "Save button not second");
      }
    );
  };
};
```

The following test case checks that the attribute `maxLength` is set on initial render and rerender.

```
testMaxLength:{
  attributes : {maxLength:"10", value:"1234567890extra"},
  test : function(component){
    var input = component.find("input").getElement();
    $A.test.assertEquals("10", input.getAttribute("maxLength"), "unexpected maxLength");

    component.set("v.maxLength", "5");
    $A.renderingService.rerender(component);

    $A.test.assertEquals("5", input.getAttribute("maxLength"), "unexpected maxLength
after rerender");
  }
},
```

The following test case checks that the specified class is set on initial render and rerender.

```
testClass:{
  attributes : {"class":"initial"},
  test : function(component){
    var div = component.find("bg").getElement();
    $A.test.assertTrue($A.util.hasClass(div, "initial"), "class not set");

    component.set("v.class", "first");
    $A.renderingService.rerender(component);

    $A.test.assertFalse($A.util.hasClass(div, "initial"), "original class not removed
after rerender");
    $A.test.assertTrue($A.util.hasClass(div, "first"), "new class not set after
rerender");
  }
},
```

### See Also:

[JavaScript Test Suite Setup](#)

## Mocking Java Classes

Use mocking to isolate your JavaScript test from other resources, such as a Java model, provider, or server-side controller. This enables you to narrow the focus of the test and eliminate other modes of failure, such as network errors. You should test the external resources in separate tests.

Aura enables you to mock a Java model, provider, or server-side controller by using a `mocks` element in your test function. `mocks` is an array of objects representing the resource that you're mocking.

Let's look at the high-level structure of a test using a mocked object. `mocks` contains `type`, `stubs`, and `descriptor` elements.

```
testSampleSyntax : {
  mocks : [{
    type : "MODEL|PROVIDER|ACTION",
    // descriptor is optional
    descriptor : ...,
    stubs : [{
      // method is optional for a model or provider
      method : { ... },
      answers : [{
        // specify value or error but not both
        value : ...
        error : ...
      }]
    }]
  }],
  test : function(cmp) {
    // test code goes here
  }
},
```

### **type**

The type of mock object. Valid values are: `MODEL`, `PROVIDER`, and `ACTION`.

### **stubs**

An array of objects representing the Java methods of the class being mocked. A stub object has `method` and `answers` properties.

### **method**

The `method` property is optional, except for the `ACTION` type. It defaults to `provide` for a provider, and `newInstance` for a model.

A method has the following elements:

- `name` is the method name.
- `params` is an array of Strings representing the input parameter types, if there are parameters.
- `type` is the return type. The default value is `Object`.

For example, this `method` element mocks `String doSomeWork(Boolean immediate, MyCustomType toProcess)`.

```
method : {
  name : "doSomeWork",
  type : "java.lang.String",
  params : ["java.lang.Boolean", "my.package.MyCustomType"]
}
```

**answers**

The `answers` property is an array of answer objects returned by the stub when it is invoked.

An answer object has either a `value` or an `error` property. This indicates whether the mock returns the given value or throws a Java exception.

The format of the `value` object depends on the class being mocked. Provider values correspond to the `ComponentConfig` object returned by `provide()`, and can specify either `descriptor` or `attributes` or both.



**Note:** The framework doesn't support custom values, such as types that require a custom converter.

Multiple answers enable you to test sequencing or multiple invocations of an action. For example, if a test simulates clicking a button twice, this would call a server action twice, and you may want the actions to return different responses.

Alternatively, your component might load two or more input fields and you want the model to return different values for each field. If the mock is invoked more times than you have answers for, the last answer is repeated. For example, if the mock for an input field value returns the answers "anybody" and "there", but the component has four input fields, the mock returns "anybody", "there", "there", "there".

The `error` property is a `String` containing the fully qualified class name of the exception thrown. You can only use exceptions with no-argument constructors, or a constructor accepting a `String`.

**descriptor**

The `descriptor` element is optional and defaults to the descriptor for the resource being mocked. For example, this is the descriptor for a model class.

```
descriptor : "java://org.auraframework.docsample.SampleJavaModel",
```

To mock the type of a super or child component, such as a child `ui:input` component, you need to specify a `descriptor`.



**Note:** The descriptor for the `ACTION` type is the controller descriptor rather than the action descriptor. For example:

```
descriptor : "java://org.auraframework.docsample.SampleJavaController",
```

**Mocking Java Models****Mocking Java Providers****Mocking Java Actions****Mocking Java Models**

This test mocks a Java model. The test function is a placeholder. You would add actual test code here.

```
testModelProperties : {
  mocks : [{
    type : "MODEL",
    stubs : [{
      answers : [{
        value : {
          secret : { value : "<not available>" } ,
          integer : { value : 1 },
          stringList : { value : [ "early", "on", "time", "late" ] }
        }
      ]
    }
  ]
}
```

```

    ]]
  }],
  test : function(cmp) {
    // test code goes here
  }
},

```

This test has a mock object that throws an exception.

```

testModelThrowsException : {
  mocks : [{
    type : "MODEL",
    stubs : [{
      answers : [{
        error : "org.auraframework.throwable.AuraRuntimeException"
      }]
    }]
  }],
  test : function(cmp) {
    // test code goes here
  }
},

```

### See Also:

[Java Models](#)  
[Mocking Java Providers](#)  
[Mocking Java Actions](#)  
[Mocking Java Classes](#)

## Mocking Java Providers

This test mocks a Java provider. The test function is a placeholder. You would add actual test code here.

```

testProviderDescriptorAndAttributes : {
  mocks : [{
    type : "PROVIDER",
    stubs : [{
      answers : [{
        value : {
          descriptor : "aura:text",
          attributes : { value : "fresh" }
        }
      }]
    }]
  }],
  test : function(cmp) {
    // test code goes here
  }
},

```

The value element for a provider corresponds to the `ComponentConfig` object returned by `provide()`, and can specify either `descriptor` or `attributes` or both.

**See Also:**

[Server-Side Runtime Binding of Components](#)

[Mocking Java Models](#)

[Mocking Java Actions](#)

[Mocking Java Classes](#)

## Mocking Java Actions

This test mocks an action in a Java server-side controller. The test function is a placeholder. You would add actual test code here.

```
testActionString : {
  mocks : [{
    type : "ACTION",
    stubs : [{
      method : { name : "getString" },
      answers : [{
        value : "what I expected"
      }]
    }]
  }],
  test : function(cmp) {
    // test code goes here
  }
},
```

This test has a mock object that throws an exception.

```
testModelThrowsException : {
  mocks : [{
    type : "ACTION",
    stubs : [{
      method : { name : "getString" },
      answers : [{
        error : "java.lang.IllegalStateException"
      }]
    }]
  }],
  test : function(cmp) {
    // test code goes here
  }
}
```

**See Also:**

[Creating Server-Side Logic with Controllers](#)

[Mocking Java Models](#)

[Mocking Java Providers](#)

## Chapter 22

### Customizing Behavior with Modes

---

#### In this chapter ...

- [Modes Reference](#)
- [Controlling Available Modes](#)
- [Setting the Default Mode](#)
- [Setting the Mode for a Request](#)

Modes are used to customize Aura framework behavior. For example, the framework is optimized for performance in `PROD` (production) mode, and ease of debugging in `DEV` (development) mode.

## Modes Reference

Aura supports different modes, which are useful depending on whether you are developing, testing, or running code in production. The list of modes in Aura is defined in the `AuraContext` Java interface.

Every request in Aura is associated with a context. After initial loading of an app, each subsequent request is an XHR POST that contains your Aura context configuration, which includes the mode to run in, and the name of the app.

We split the list of modes into two sections here to differentiate between runtime and test modes. This split is purely to cluster similar modes together in the documentation. All the runtime and core modes are defined in the `Mode` enum in `AuraContext`.

All modes are available by default in your app. Many of the modes use the Google Closure Compiler, which is a tool for optimizing JavaScript code.

### Runtime Modes

Use these modes for running in development or production.

Mode	PROD	DEV	PRODDEBUG
<b>Usage</b>	Use for apps in production. The framework is optimized for performance rather than ease of debugging in this mode.	Use for apps in development. The framework is configured for ease of debugging in this mode.	Use temporarily to debug apps in production.
<b>Debugging</b>	Not recommended for debugging.  Since <code>PROD</code> mode is intended for apps in production, test modes, such as <code>SELENIUM</code> , are preferable for running tests, especially concurrent tests.	Facilitates debugging. Pretty prints JSON responses from the server. Exposes private members in some framework JavaScript objects.	Similar to <code>PROD</code> mode
<b>Access</b>	Disables access to a <code>.cmp</code> resource in a URL. You can only access a <code>.app</code> resource.	Enables a <code>.cmp</code> resource to be addressed in a URL.	Similar to <code>PROD</code> mode
<b>Google Closure Compiler</b>	Uses the Google Closure Compiler to optimize the JavaScript code. The method names and code are heavily obfuscated.	Uses the Google Closure Compiler to lightly obfuscate the names of non-exported JavaScript methods. This is meant to avoid unintentional usage of non-exported methods.	Does not use Google Closure Compiler
<b>Caching</b>	Caches code. When a file change is detected, this mode performs a full closure compile on all units.	Caches code. When a file change is detected, this mode clears the cache and recompiles definitions.	Similar to <code>PROD</code> mode

### Test Modes

Use these modes for running different flavors of tests. The various test modes mainly expose extra JavaScript calls that are not available in runtime modes.

In all test modes, caching of registries between tests is disabled. If you modify a cached definition in a test, the modified cached definition is not visible to subsequent tests.

Mode	Usage
JSTEST	<p>Use for running component tests. If your component or app has a <code>&lt;componentName&gt;Test.js</code> file in its bundle, a browser page is displayed to run the tests. A tab is displayed for each test case in your test suite. Each tab contains an iframe that loads the component in <code>AUTOJSTEST</code> mode and runs the single test case.</p> <p>The test results are displayed below the iframe. For a successful test run, the tab turns green; for a failure, it turns red.</p>
JSTESTDEBUG	Use for debugging component tests. Similar to <code>JSTEST</code> mode but doesn't use the Google Closure Compiler.
AUTOJSTEST	<p>Used by <code>JSTEST</code> mode when running inside the iframe for a test case. It enables extra JavaScript needed to execute the test case.</p> <p>Use this mode by requesting the component or app containing the test in <code>JSTEST</code> mode.</p>
AUTOJSTESTDEBUG	<p>Used by <code>JSTESTDEBUG</code> mode when running inside the iframe for a test case. It enables extra JavaScript needed to execute the test case.</p> <p>Use this mode by requesting the component or app containing the test in <code>JSTESTDEBUG</code> mode.</p>
PTEST	<p>Use for running performance tests using the Jiffy Graph UI. Loads Jiffy performance test tools and enables the Jiffy Graph UI. Jiffy is an end-to-end real-world web page instrumentation and measurement suite.</p> <p>This mode doesn't use the Google Closure Compiler.</p>
CADENCE	<p>Use for running performance tests if you want to use Jiffy metrics and track the numbers server-side. Loads and runs Jiffy performance test tools and logs the results on the server.</p> <p>Cadence tests use Jiffy, but don't load the Jiffy Graph UI.</p>
SELENIUM	Use for tests with Selenium, a software testing framework for web apps. This mode uses the Google Closure Compiler.
SELENIUMDEBUG	Similar to <code>SELENIUM</code> mode but doesn't use the Google Closure Compiler.
UTEST	Used for running unit tests against the framework. It allows developers of the framework to enable some debug code only during testing.
FTEST	Similar to <code>UTEST</code> mode, but used for functional tests instead of unit tests. This mode may expose different debug code than <code>UTEST</code> mode.
STATS	Used for compiling statistics for use with the query language.

### See Also:

[Component Bundles](#)  
[Setting the Default Mode](#)  
[Testing Components](#)



## Controlling Available Modes

You can customize the set of available modes in your application by writing a Java class that implements the `getAvailableModes()` method in the `ConfigAdapter` interface. The default implementation in `ConfigAdapterImpl` makes all modes available.

So, if you want to use your own configuration to limit the modes in certain environments, such as a production environment, you could limit the modes to only allow `PROD` mode. This would ensure that `PROD` mode is used for all requests. The default mode is not used if it's not also included in the list of available modes.

### See Also:

[Modes Reference](#)

[Setting the Default Mode](#)

[Setting the Mode for a Request](#)

## Setting the Default Mode

The default mode is `DEV`. This is defined in the `ConfigAdapterImpl` Java class.

You can change the default mode to `PROD` by setting the `aura.production` Java system property to `true`. Do this by adding `-Daura.production=true` to the arguments when you are starting your server.

To set an alternate default mode, write a Java class that implements the `getDefaultMode()` method in the `ConfigAdapter` Java interface.

The default mode is not used if it's not also included in the list of available modes.

### See Also:

[Controlling Available Modes](#)

[Setting the Mode for a Request](#)

[Modes Reference](#)

## Setting the Mode for a Request

Each application has a default mode, but you can change the mode for each HTTP request by setting the `aura.mode` parameter in the query string. If the requested mode is in the list of available modes, the response for that mode is returned. Otherwise, the default mode is used.

For example, let's assume that `DEV` and `PROD` are in the set of the available modes. If the default mode is `DEV` and you want to see the response in `PROD` mode, use `aura.mode=PROD` in the query string of the request URL. For example:

```
http://<your server>/demo/test.app?aura.mode=PROD
```

### See Also:

[Modes Reference](#)

[Setting the Default Mode](#)

[Controlling Available Modes](#)

[URL-Centric Navigation](#)

# Chapter 23

## Debugging

---

### In this chapter ...

- [Log Messages](#)
- [Warning Messages](#)
- [Debugging with Network Traffic](#)
- [Aura Debug Tool](#)
- [Querying State and Statistics](#)

There are several tools and techniques that can help you to debug Aura applications.

## Log Messages

To help debug your client-side Aura code, you can use the `log()` method to write output to the JavaScript console of your web browser.

Use the `$A.log(string, [error])` method to output a log message to the JavaScript console. The first parameter is the string to log and the optional second parameter is an error object whose messages should be logged. For example, `$A.log("This is a log message");` will output "This is a log message" to the JavaScript console. If you put `$A.log("The name of the action is: " + this.getDef().getName());` inside an action called "openNote" in a client-side controller, then the log message "The name of the action is: openNote" will be output to the JavaScript console. The output is also sent to the Aura Debug Tool.

For instructions on using the JavaScript console, refer to the instructions for your web browser.

## Warning Messages

To help debug your client-side Aura code, you can use the `warning()` method to write output to the JavaScript console of your web browser.

Use the `$A.warning(string)` method to write a warning message to the JavaScript console. The parameter is the message to display. For example, `$A.warning("This is a warning message.");` will output "This is a warning message." to the JavaScript console. A stack trace will also be displayed in the JavaScript console. The output is also sent to the Aura Debug Tool.

For instructions on using the JavaScript console, refer to the instructions for your web browser.

## Debugging with Network Traffic

Looking at JSON network traffic can help you identify performance hotspots and tune your app.



**Note:** This topic describes an internal wire protocol that is subject to change at any time.

The Google Chrome Developer Tools let you look at JSON messages on the wire as they travel between the client and the server. The JSON messages are structured in a way that is readable. They are not binary encoded. For example, you can see when a `componentDef` is coming across or you could look for data and metadata going across the wire when the metadata should actually be cached. If metadata is repeatedly being sent across the wire, this can have a severe impact on performance.

In this tutorial, we will use the Aura Note sample application to illustrate how JSON messages can be viewed.

1. Get the latest version of the sample application by typing `git clone https://github.com/forcedotcom/aura-note.git` at a command prompt.
2. Type `cd aura-note`
3. Type `mvn jetty:run -Pdev`
4. In Google Chrome, browse to `http://localhost:8080/auranote/notes.app`
5. Right click on the Aura Note web page and select **Inspect Element** to open the Chrome Developer Tools.
6. In the Chrome Developer Tools window, click the **Network** tab.
7. In the Note Title field in Aura Note, type `My Test Note`. In the text field, type `This is my note text` and click **Save**.

## Understanding the Request

In the Chrome Developer Tools **Network** tab, two XMLHttpRequest (XHR) requests are displayed. For an explanation of what each column means, refer to the Chrome Developer Tools documentation.

Click the first request and then click the **Headers** tab. The **Headers** tab displays the request that is being sent to the server. In the **Form Data** section, you can see the message and `aura.context`:

```
Form Data
message:
{"actions":[{"id": "44.2",
  "descriptor":
    "java://org.auraframework.demo.notes.controllers.NoteEditController/ACTION$saveNote",
    "params": {
      "title": "My Test Note",
      "body": "This is my note text",
      "latitude": null,
      "longitude": null
    }
}]}
aura.context: {
  "mode": "DEV",
  "loaded": {"APPLICATION@markup://auranote:notes": "BUI3ODiAK-fwLFsL70ufNQ"},
  "app": "auranote:notes",
  "lastmod": "1376946254000",
  "fwuid": "ZWE1dXJKa3FSSWtQSzZ6S2NtSWdzQQ"
}
aura.num:
2
```

### aura.context

The `aura.context` is the JSON encoded information that we need to send up to the server every time we communicate with it. It tells us some basic information about the client.

- `mode`: Describes the runtime mode that the client is operating in.
- `loaded`: Tells the server about the application and the version of the client. For example, `"loaded": {"APPLICATION@markup://auranote:notes": "BUI3ODiAK-fwLFsL70ufNQ"}` means that the application is in the `auranote` namespace, the application is `notes` and the version unique hash is `"BUI3ODiAK-fwLFsL70ufNQ"`. The version is used for change detection so the client can be updated to a new version when necessary. Optionally, if any components are dynamically loaded, they will be displayed at the end of the list. In this example, there is nothing to display. Here is an example from an app that has a component called `tutorialsNav` that is loaded dynamically:

```
"loaded" : {
  "APPLICATION@markup://auradocs:docs" : "4df774xhioeHpDZrhT4cuQ",
  "COMPONENT@markup://auradocs:tutorialsNav" : "Lt2UR0hShMLcZT0845WSaA"
}
```

- `fwuid`: The framework unique id is a hash that is used as a fingerprint to detect if the framework has changed.

### aura.num

The `aura.num` displays the number of XHRs that the client has made to the server. This is so we can ensure that any component global IDs contain the `aura.num` as their suffix. The `aura.num` attribute gets reset to 0 when you refresh the browser because it is only valid for the life of the page. If you navigate away from an Aura app, the JavaScript memory space gets cleared and the `aura.context` and all related data is destroyed. The `aura.num` attribute guarantees that global IDs will always be unique, even when they are created on the server. This is useful because Aura lets you do incremental, partial-page updates.

**message**

The message contains one or more actions that describe what to do at the server.

- **id:** The id of the action. The postfix of the id is the `aura.num`. It will be unique across the lifetime of an `aura.context`. This id is useful when the response is returned because we can use it to look up the callback, if there is one, and complete the processing of the round trip.
- **descriptor:** The descriptor for the action. In the example, the descriptor is on the `NoteEditController` in Java and it is the `saveNote` action.
- **params:** The parameters that get sent to the server. In the example, you can see the title and the body of the note that we created.

If a problem is happening at this point or if performance is slow, add a breakpoint in `saveNote` in `NoteEditController` to investigate. It lets you know where you could put a breakpoint on the server side. It is not uncommon to see several actions in one message being sent in one trip to the server. Actions are run one after the other so one slow action could slow your whole app down. Looking at this type of response can help you figure out which one is causing the problems.

**Understanding the Response**

1. In the Chrome Developer Tools window, click the **Response** tab to see the raw response that is being sent back from the server.
2. Next, click the **Preview** tab. It displays a formatted view of the response.

The context section of the response will look something like this:

```
context: {mode:DEV, app:auranote:notes, requestedLocales:[en_US, en],...}
  app: "auranote:notes"
  fwuid: "ZWE1dXJKa3FSSWtQSzZ6S2NtSWdzQQ"
  globalValueProviders: [{type:$Browser,...},...]
    0: {type:$Browser,...}
      type: "$Browser"
      values: {formFactor:DESKTOP, isWindowsPhone:false, isPhone:false,
isFIREFOX:false, isIPad:false,...}
      1: {type:$Locale, values:{language:en, country:US, variant:, langLocale:en_US,
dateFormat:MMM d, yyyy,...}}
      type: "$Locale"
      values: {language:en, country:US, variant:, langLocale:en_US, dateFormat:MMM
d, yyyy,...}
    lastmod: "1376946254000"
    loaded: {APPLICATION@markup://auranote:notes:BUI3ODiAK-fwLFsL70ufNQ}
    mode: "DEV"
    requestedLocales: [en_US, en]
```

It is very similar to what was sent to the server in the request. The context has been updated and it might have some items added to it. In this example, now there are `globalValueProviders`. The action we just ran may cause us to go get more labels from the server. They come back in the `globalValueProviders` which is the global state. It is not specific to any action. The `loaded` list may have been updated as well. In this example, it went up and came back the same. However, if a component had been created on the server side then that would show up in the `loaded` list. This is for lazy-loading of metadata, which is very important from a performance standpoint.

The actions section of the response will look something like this:

```
actions: [{id:44.2, state:SUCCESS,...}]
  0: {id:44.2, state:SUCCESS,...}
    error: []
    id: "44.2"
    returnValue: {id:5, title:My Test Note, body:This is my note text,
createdOn:2013-08-19T21:15:28.062Z}
    state: "SUCCESS"
```

This is the response for the action that we sent out in the request. You can see that there were no errors. The id of the action is echoed back so that callbacks can be looked up so Aura can call the callback and pass it this return value. Latitude and longitude are not included in the `returnValue` because in this example they are set to null and by default, the JSON serializer will omit null values. The possible values for `action.state` are `SUCCESS`, `FAILURE` and `INCOMPLETE`. `INCOMPLETE` means that the server couldn't be reached due to connectivity issues or the action was marked as abortable and new actions were pushed into the action queue before one or more abortable actions completed.

Earlier, we examined the XHR request in the **Headers** tab and saw that the `saveNote` action of the `NoteEditController` was being called. Thus, we can open the `NoteEditController.java` file in the `aura-note` sample application and look at the implementation of the `saveNote` function and observe that it returns a `Note` Java object. Next, if we look at `Note.java`, we see that the `Note` object is serializable and there is a method which creates the JSON payload that is eventually put into the action's `returnValue`. Both the client and the server side know that `saveNote` returns a `Note` and we will take this return value and turn it into something that looks like a note on the JavaScript side.

## Understanding the Second Request

Click the second request in the **Network** tab and then click the **Headers** tab.

In the **Form Data** section, you can see that the descriptor of the action is on the `ComponentController` and is the `getComponent` action. This action is to get an instance of `auranote:noteList` and has no attributes. This action is for refreshing the list of notes on the left side of the app.

```
message:
{"actions":[{"id":"158.9","descriptor":"aura://ComponentController/ACTION$getComponent",
"params":{"name":"markup://auranote:noteList","attributes":{}}]}
```

## Understanding the Second Response

Click the **Preview** tab to view the server's response.

```
actions: [{id:158.9, state:SUCCESS, returnValue:{serId:1,...}, error:[],
components:{1:158.9:{serRefId:1}}}]
0: {id:158.9, state:SUCCESS, returnValue:{serId:1,...}, error:[],
components:{1:158.9:{serRefId:1}}
  components: {1:158.9:{serRefId:1}}
  error: []
  id: "158.9"
  returnValue: {serId:1,...}
    serId: 1
    value: {componentDef:{serId:2, value:{descriptor:markup://auranote:noteList}},
globalId:1:158.9,...}
    attributes: {serId:3, value:{values:{sort:createdOn.desc}}}
    componentDef: {serId:2, value:{descriptor:markup://auranote:noteList}}
    globalId: "1:158.9"
    model: {notes:[{id:7, title:My Test Note, body:This is my note text,
createdOn:2013-08-20T03:35:54.034Z},...]}
      notes: [{id:7, title:My Test Note, body:This is my note text,
createdOn:2013-08-20T03:35:54.034Z},...]
        0: {id:7, title:My Test Note, body:This is my note text,
createdOn:2013-08-20T03:35:54.034Z}
        1: {id:4, title:My new note, body:lorem ipsum. ,
createdOn:2013-08-17T01:33:15.508Z}
        state: "SUCCESS"}
```

The `actions.returnValue.value` section tells you the `componentDef` value. In this case, it just returns the descriptor which means we already know about `noteList` on the client side so we do not need to send the metadata again. The server generated this to tell the client side to create an instance of `noteList` with the `globalId` of `1:158.9`.

Next, look at the model:

```
model: {notes:[{id:7, title:My Test Note, body:This is my note text,
createdOn:2013-08-20T03:35:54.034Z},...]}
  notes: [{id:7, title:My Test Note, body:This is my note text,
createdOn:2013-08-20T03:35:54.034Z},...]
    0: {id:7, title:My Test Note, body:This is my note text,
createdOn:2013-08-20T03:35:54.034Z}
    1: {id:4, title:My new note, body:lorem ipsum. , createdOn:2013-08-17T01:33:15.508Z}
```

You can see what the model contains because the data is in a format that is readable. You can verify that the information being returned is correct.

For performance reasons, it is important to reduce the volume of data being transferred. There is a highly redundant structure in the JSON. Many objects are referenced over and over again. So, instead of sending the same data over and over again, and bloating the size of the responses, the JSON encoder on the server side figures out which objects have already been transferred. To do this, Aura uses reference serialization for metadata and assigns a serialization ID (`serId`) to each of these objects. This means that they can be referenced later. In the **Response** tab, you can see the `serId` for the `componentDef`, which is a reference to `noteList`:

```
"actions": [
  {
    "id": "158.9",
    "state": "SUCCESS",
    "returnValue": {
      "serId": 1,
      "value": {
        "componentDef": {
          "serId": 2,
          "value": {
            "descriptor": "markup://auranote:noteList"
          }
        }
      }
    }
  }
  ...
]
```

Although the network traffic initially looks like a lot of noise, it can yield valuable information to help you identify problems and performance hotspots.

### See Also:

[Modes Reference](#)

[Abortable Actions](#)

[Testing Components](#)

## Aura Debug Tool

The Aura debug tool outputs debug information about an Aura component.



**Note:** You must disable the popup blocking feature of your web browser to use the debug tool.

It opens a separate browser window. The debug tool has the following tabs: Errors, Warnings, Components, Events, Storage, Accessibility, and Console.

To launch the Aura Debug tool, add the query string `aura.debugtool=true` after the URL of the Aura component file that you are viewing in your browser. For example:

```
http://localhost:8080/auranote/noteList.cmp?aura.debugtool=true
```

To display additional statistics in the Components tab, append the query string `aura.mode=STATS` to the URL. For example:

```
http://localhost:8080/auranote/noteList.cmp?aura.debugtool=true&aura.mode=STATS
```

### See Also:

[Modes Reference](#)

[Testing Components](#)

## Querying State and Statistics

To aid debugging and testing, you can use Aura's query language to see the current state of certain objects in a running app. The query language is available in your browser's console for all modes, except for `PROD` mode.

You can get extra statistics about the app by running queries in `STATS` mode. This can help with performance tuning.

### Viewing Help for Command-Line Options

To get usage instructions for the query language, run this command in your browser's console:

```
$A.qhelp()
```

### Querying All Components

To query all components on a page, run:

```
$A.getQueryStatement().query()
```

Expand the `ResultSet` to drill into the components and their details. This query can return many components. To find the type of one of the components returned in the `rows` array, call `toString()`. For example, to get the type of the first returned component, run:

```
$A.getQueryStatement().query().rows[0].toString()
```

### Selecting Fields

The default is to return all fields, but you can be more selective by using `field()`. For example, to return a few fields, run:

```
$A.getQueryStatement().field("toString, globalId").query()
```

Use a comma-separated list of fields or chain calls to `field()`. For example, this query mixes both types of `field()` syntax.

```
$A.getQueryStatement().field("toString, globalId").field("super").field("def").query()
```

You can use an expression as a field too. For example, to get the value of an attribute called `description`, run:

```
$A.getQueryStatement().field("toString, v.description").query()
```

If a field name doesn't match, the query engine also uses `get` and `is` prefixes to resolve function names. So, you can use the same syntax as your markup to access a field in your model, such as `m.firstName` to match the `getFirstName` method in the model's class.



## Defining Derived Fields and Filtering

You can create a derived field by adding your own logic to process the fields available in the view that you're querying. Derive your own fields by using `field("derivedFieldName", "derivedFieldMethodChain")`. For example:

```
$A.getQueryStatement().field("descriptor", "getDef().getDescriptor().toString()").query()
```

Derived fields are particularly useful when you want to filter the query results using `where()`. For example:

```
$A.getQueryStatement().field("descriptor",
"getDef().getDescriptor().toString()").where("descriptor ==
'markup://aura:application'").query()
```

## Choosing a View

All the queries so far have looked at components, but you can use `from()` to explore other views, such as `componentDef`. For example:

```
$A.getQueryStatement().from("componentDef").query()
```

If the query doesn't include `from()`, the default is the `component` view.

To get a list of available views, run:

```
$A.devToolService.views
```

Use the `STATS` mode to see extra views for value objects.

## Querying Value Objects

You can query value objects in `STATS` mode. For example:

```
$A.getQueryStatement().from("value").field("toString").query()
```

## Grouping by Fields

Use `groupBy` to group your results by a field. For example, to group by the different types of value objects, run:

```
$A.getQueryStatement().from("value").field("toString").groupBy("toString").query()
```

## Diffing Query Result Sets

To get a diff between two result sets, use `diff()`. For example:

```
var before = $A.getQueryStatement().query(); var after = $A.getQueryStatement().query();
after.diff(before);
```

This is useful if you want to perform operations between running the `before` and `after` queries and analyze the diff between the two result sets.

### Plugging in Custom Code with Adapters

---

#### In this chapter ...

- [Default Adapters](#)
- [Overriding Default Adapters](#)

Aura has a set of adapters that provide default implementations of functionality that you can override.

For example, the localization adapter provides the default behavior for working with labels and locales. You may want to override this behavior for your own localization requirements.

Think of an adapter as a plugin point for your custom code. It's useful to contrast this with the Aura Integration Service, which enables you to inject Aura components into a Web app that is not developed in Aura.

`AuraAdapter` is the base marker interface for all adapters. You can find all the adapter interfaces in the `org.auraframework.adapter` package.

#### See Also:

[Default Adapters](#)

[Overriding Default Adapters](#)

[Accessing Components from Non-Aura Containers](#)

## Default Adapters

Aura has a set of default adapters.

Adapter	Description
ComponentLocationAdapter	Provides the default location for storing component source files. The default is to store components on the filesystem but you could override this to store them in a database.
ConfigAdapter	Provides many defaults, including the set of available modes, and the version of the Aura framework.
ContextAdapter	Provides the default context. Every request in Aura is associated with a context. After initial loading of an app, each subsequent request is an XHR POST that contains your Aura context configuration, which includes the mode to run in, the name of the app, and the namespaces that already have metadata loaded on the client.
ExceptionHandler	Provides the default exception handling. The default is to log the exception.
ExpressionAdapter	Provides the default expression language.
FormatAdapter	Provides the default implementations for reading and writing different resources, such as Aura markup, CSS, or JSON.
GlobalValueProviderAdapter	Provides the global value providers. Global value providers are global values, such as <code>\$Label</code> , that a component can use in expressions.
JsonSerializerAdapter	Provides the default JSON serializers. You can use this adapter to customize how Aura locates the correct serializer implementation to marshall objects to and from JSON.
LocalizationAdapter	Provides the default label and locale handling.
LoggingAdapter	Provides the default logging.
PrefixDefaultsAdapter	Provides the default prefixes for Aura definitions. Each definition describes metadata for an element, such as a component, event, controller, or model.
RegistryAdapter	Provides the default registries. Registries store metadata definitions. Some registries last for the duration of a request, while others are cached for the lifetime of an app.
StyleAdapter	Provides the default CSS themes.

### See Also:

[Plugging in Custom Code with Adapters](#)  
[Overriding Default Adapters](#)

## Overriding Default Adapters

There are several ways to override the default adapters.

To override one of the default adapters:

1. Extend an existing adapter or create a new class that implements the adapter interface that you're overriding.

2. Use the `@Override` annotation on each interface method that you implement.

**See Also:**

*[Plugging in Custom Code with Adapters](#)*

*[Default Adapters](#)*

*[Customizing your Label Implementation](#)*

## Chapter 25

### Accessing Components from Non-Aura Containers

---

#### In this chapter ...

- [Add an Aura button inside an HTML div container](#)

The Aura Integration Service enables plugging Aura components into non-Aura HTML containers.

Because Aura requires an app to start and to render components, the Aura Integration Service creates and manages an internal integration app on your behalf for the components you're embedding. This makes it easy to use Aura components in an HTML-based application.

Also, the Aura Integration Service allows partial page updates. You can add additional components to a page that has already been loaded and after an app has already been created.

An Aura component instance is embedded in a page inside a script tag and is bound to its parent DOM element.

The Aura Integration Service provides a set of Java APIs that allow you to embed a component. The Java APIs are included in the following interfaces and their class implementations.

- **IntegrationService Interface** (implemented by `IntegrationServiceImpl`): Enables the creation of an integration using the `createIntegration()` method.
- **Integration Interface** (implemented by `IntegrationImpl`): Enables adding components using the `injectComponent()` method.



**Note:** The Aura History Service and Aura Layout Service are not supported with the Aura Integration Service, and hence embedded components can't make use of these services.

#### See Also:

[Customizing Behavior with Modes](#)  
[Component IDs](#)

## Add an Aura button inside an HTML div container

The Aura Integration Service lets you plug Aura components into HTML containers.

1. Create an instance of the Aura Integration Service.

```
IntegrationService svc = Aura.getIntegrationService();
```

2. Create an integration, which allows you to embed components in your page.

For the first argument, pass the context path. For servlets in the default root context, it is an empty string. For the second argument, pass the mode. In this example, we're specifying the `DEV` mode. For the third argument, pass a Boolean value to indicate whether Aura should create an integration app or not. In this case, we're passing `true`. If you want to perform a partial page update, pass `false` for the third argument. This allows you to add more components after a page has been loaded and an app has already been created.

```
Integration integ = svc.createIntegration("", Mode.DEV, true);
```

3. Call the `injectComponent` method to embed a component in a parent container.

For the first argument, pass the component's fully qualified name. In this case, it is `"ui:button"` (`ui` is the namespace and `button` is the component's name). For the second argument, pass the component's attributes as a map. This example creates a map with one attribute and passes it as the second argument. For the third argument, pass the local component ID. In this example, it is `"button1"`. For the fourth argument, pass the DOM identifier for the parent container element. In this example, it is `"div1"`. For the fifth argument, pass a buffer that will contain the script output.

```
Map<String, Object> attributes = Maps.newHashMap();
attributes.put("label", "Click Me");
Appendable out = new StringBuffer();
integration.injectComponent("ui:button", attributes, "button1", "div1", out);
```

The following is the full listing of the sample.

```
IntegrationService svc = Aura.getIntegrationService();
Integration integration = svc.createIntegration("", Mode.DEV, true);
Map<String, Object> attributes = Maps.newHashMap();
attributes.put("label", "Click Me");
Appendable out = new StringBuffer();
integration.injectComponent("ui:button", attributes, "button1", "div1", out);
```

## Chapter 26

### Customizing Data Type Conversions

---

#### In this chapter ...

- [Registering Custom Converters](#)
- [Custom Converters](#)

A custom converter enables the conversion of one Java type to another Java type for client data sent to the server or for server markup data.

When a client calls a server-side controller action, data that the client sends, such as input parameters for a server action, is sent in JSON format. The JSON representation of data is converted to target Java types on the server. Similarly, values in Aura markup on the server, such as component attribute values, are evaluated as Java strings. These strings are converted to corresponding Java types. For primitive Java types, the type conversion is implicit and doesn't require the addition of any converters. For example, a JSON string is converted to a Java string, and a JSON list is converted to a Java ArrayList. For custom types, or when there is no one-to-one mapping between the source value and the target type, Aura calls the custom converter that you provide to create an instance of the custom Java type corresponding to the JSON representation on the client or the markup attribute value on the server.

An example of a custom converter is a converter used to convert comma-delimited string values to an ArrayList. A component attribute of type List can have a default value in markup of a comma-delimited string of values. Aura converts this attribute string value into an ArrayList by calling the custom String to ArrayList converter.

#### See Also:

[Custom Java Class Types](#)

[Creating Server-Side Logic with Controllers](#)

[Supported aura:attribute Types](#)

## Registering Custom Converters

Register a custom converter to enable conversion of one Java type to another Java type when sending data to and from the server.

To register a custom converter:

1. Create a class that implements the `Converter` interface. Add `implements Converter<Type1, Type2>` at the end of the first line of your class definition, after the class name. Replace `Type1` with the original Java type and `Type2` with the target Java type. Next, implement each method in the `Converter` interface. For better readability of your code, we recommend you name the class using the format `Type1ToType2Converter`. This is an example of a skeletal class implementing the `Converter` interface. `Type1` and `Type2` are placeholders for the Java original type and the converted type, respectively.

```
public class Type1ToType2Converter implements Converter<Type1, Type2> {

    @Override
    public Type2 convert(Type1 value) {
        // Convert value into a value of Type2 and return it.
        // Return converted value.
    }

    @Override
    public Class<Type1> getFrom() {
        // return Type1.class;
    }

    @Override
    public Class<Type2> getTo() {
        // return Type2.class;
    }

    @Override
    public Class<?>[] getToParameters() {
        // Return the types contained in the custom type.
    }

}
```

2. Create another class annotated with `@AuraConfiguration`. The class must be in the configuration package.
3. Add a `public static` method to this class annotated with `@Impl`. The method should return either the `Converter<?, ?>` type or `Converter<Type1, Type2>` with the actual original and target Java types. The method returns a new instance of the class you created earlier, which implements the `Converter` interface.

```
package configuration;

@AuraConfiguration
public class MyTypeConverterConfig {
    @Impl
    public static Converter<Type1, Type2> exampleTypeConverter() {
        return new Type1ToType2Converter();
    }
}
```

4. To specify additional conversions, repeat the previous steps. Each new conversion requires a converter implementation class and the addition of a corresponding method to the Aura configuration class.



## Custom Converters

Here are a few examples of custom converters.

### Example 1: Custom Type Conversion for a Component Attribute

This example shows how to add a converter to convert an attribute string value to the corresponding custom type. It contains the definition of the custom type, `MyCustomType`, an example of the attribute, the corresponding converter, and a method in the Aura configuration class.

This is the definition of the custom type, `MyCustomType`.

```
package doc.sample;

public class MyCustomType implements JsonSerializer {
    private String val;

    public MyCustomType(String val) {
        this.val = val;
    }

    @Override
    public void serialize(Json json) throws IOException {
        json.writeString(val);
    }
}
```

This is the attribute of type `MyCustomType` with a default value of "x".

```
<aura:attribute name="myObj" type="java://doc.sample.MyCustomType" default="x"/>
```

This is the converter implementation for converting a string (the attribute value) to an object of type `MyCustomType` (the target Java type).

```
public class StringToMyCustomTypeConverter implements Converter<String, MyCustomType> {

    @Override
    public MyCustomType convert(String value) {
        return new MyCustomType(value);
    }

    @Override
    public Class<String> getFrom() {
        return String.class;
    }

    @Override
    public Class<MyCustomType> getTo() {
        return MyCustomType.class;
    }

    @Override
    public Class<?>[] getToParameters() {
        return null;
    }
}
```

This is the corresponding Aura Configuration method.

```
package configuration;

@AuraConfiguration
public class MyCustomTypeConverterConfig {
    @Impl
    public static Converter<String, MyCustomType> exampleTypeConverter() {
        return new StringToMyCustomTypeConverter();
    }
}
```

## Example 2: Parameterized Type Conversion for a Server Action Call

This example shows how to add a converter to convert the type of a parameter passed to a server-side controller action call that a client makes. The target type of the conversion is a parameterized type, `List<MyCustomType>`, which is a list of `MyCustomType` objects.

This example is based on the `MyCustomType` class defined earlier.

This is the client call to the `accept` action on the server-side controller. The client passes an array of three string values that corresponds to a list of `MyCustomType` objects. Because the parameter value is an array of objects, the original type of the conversion is `ArrayList`.

```
custom : function(c) {
    var a = c.get("c.accept");
    a.setParams({myObjs: ["x", "y", "z"]});
    $A.enqueueAction(a);
},
```

This is how the `accept` method looks in the server-side controller. Notice the parameter of the `accept` method is of type `List<MyCustomType>`. This is the target type of the conversion.

```
@AuraEnabled
public static void accept(@Key("myObjs") List<MyCustomType> myObjs) {
    for (MyCustomType obj : myObjs) {
        System.err.println("MyCustomType:" + obj);
    }
}
```

This is the converter implementation that converts an `ArrayList` (the parameter array sent by the client) to a `List` of `MyCustomType` objects on the server.

```
public class ArrayListToMyCustomTypeListConverter implements Converter<ArrayList, List> {

    @Override
    public List<MyCustomType> convert(ArrayList value) {
        List<MyCustomType> retList = Lists.newLinkedList();
        for (Object part : value) {
            retList.add(new MyCustomType(part.toString()));
        }
        return retList;
    }

    @Override
    public Class<ArrayList> getFrom() {
        return ArrayList.class;
    }

    @Override
    public Class<List> getTo() {
        return List.class;
    }
}
```

```

    }

    @Override
    public Class<?>[] getToParameters() {
        return new Class[] { MyCustomType.class };
    }
}

```

This is the corresponding Aura Configuration method.

```

package configuration;

@AuraConfiguration
public class MyCustomTypeListConverterConfig {
    @Impl
    public static Converter<ArrayList, List<MyCustomType>> exampleTypeConverter() {
        return new ArrayListToList<MyCustomType>Converter();
    }
}

```

### Example 3: Parameterized Type Conversion for a Component Attribute

This example is similar to the previous one except that the conversion is done for an attribute value. In this example, consider the following attribute that holds a list of `MyCustomType` objects and with a default value of `"x,y,z"`. Because the attribute value is a string, the original type of the conversion is `String`. The target type is `List<MyCustomType>`.

This example is based on the `MyCustomType` class defined earlier.

```

<aura:attribute name="myObjs" type="java://java.util.List<doc.sample.MyCustomType>"
default="x,y,z"/>

```

This is the converter implementation for converting a string to a list of `MyCustomType` objects.

```

public class StringToMyCustomTypeListConverter implements Converter<String, List> {

    @Override
    public List<MyCustomType> convert(String value) {
        List<MyCustomType> retList = Lists.newLinkedList();
        for (String part : AuraTextUtil.splitSimple(",", value)) {
            retList.add(new MyCustomType(part));
        }
        return retList;
    }

    @Override
    public Class<String> getFrom() {
        return String.class;
    }

    @Override
    public Class<List> getTo() {
        return List.class;
    }

    @Override
    public Class<?>[] getToParameters() {
        return new Class[] { MyCustomType.class };
    }
}

```

This is the corresponding Aura Configuration method.

```
package configuration;

@AuraConfiguration
public class MyCustomTypeList2ConverterConfig {
    @Impl
    public static Converter<String, List<MyCustomType>> exampleTypeConverter() {
        return new StringToList<MyCustomType>Converter();
    }
}
```

# REFERENCE

## Chapter 27

### Reference Overview

---

#### In this chapter ...

- [aura:application](#)
- [aura:component](#)
- [aura:clientLibrary](#)
- [aura:dependency](#)
- [aura:event](#)
- [aura:if](#)
- [aura:interface](#)
- [aura:iteration](#)
- [aura:renderIf](#)
- [aura:set](#)
- [Supported HTML Tags](#)
- [Supported aura:attribute Types](#)

This section contains reference documentation including details of the various tags available in Aura.

The [Reference tab](#) includes more reference information, including descriptions and source for the components that come out-of-the-box with Aura.

## aura:application

An Aura app is a special top-level component whose markup is in a `.app` resource.

The markup looks similar to HTML and can contain Aura 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
<code>access</code>	String	Indicates whether the app can be extended by another app outside of a namespace. Possible values are <code>internal</code> (default), <code>public</code> , and <code>global</code> .
<code>controller</code>	String	The server-side controller class for the app. The format is <code>java://&lt;package.class&gt;</code> .
<code>description</code>	String	A brief description of the app.
<code>extends</code>	Component	The app to be extended, if applicable. For example, <code>extends="namespace:yourApp"</code> .
<code>extensible</code>	Boolean	Indicates whether the app is extensible by another app. Defaults to <code>false</code> .
<code>implements</code>	String	A comma-separated list of interfaces that the app implements.
<code>locationChangeEvent</code>	Event	The client-side <code>AuraHistoryService</code> monitors the location of the current window for changes. If the <code>#</code> value in a URL changes, the <code>AuraHistoryService</code> fires an Aura application event. The <code>locationChangeEvent</code> defines this event. The default value is <code>aura:locationChange</code> . The <code>locationChange</code> event has a single attribute called <code>token</code> , which is set with everything after the <code>#</code> value in the URL.
<code>model</code>	String	The model class used to initialize data for the app. The format is <code>java://&lt;package.class&gt;</code> .
<code>preload</code>	String	Deprecated. Use the <a href="#">aura:dependency</a> tag instead.  If you use the <code>preload</code> system attribute, the framework internally converts the value to <code>&lt;aura:dependency&gt;</code> tags.
<code>render</code>	String	Renders the component using client-side or server-side renderers. If not provided, Aura determines any dependencies and whether the application should be rendered client-side or server-side.  Valid options are <code>client</code> or <code>server</code> . The default is <code>auto</code> .  For example, specify <code>render="client"</code> if you want to inspect the application on the client-side during testing.
<code>renderer</code>	String	Only use this system attribute if you want to use a custom client-side or server-side renderer. If you don't set a renderer, Aura uses its default rendering, which is sufficient for most use cases. If you don't define this system attribute, your application is autowired to a client-side renderer named <code>&lt;appName&gt;Renderer.js</code> , if it exists in your application bundle.
<code>template</code>	Component	The name of the template used to bootstrap the loading of the Aura framework and the app. The default value is <code>aura:template</code> . You can customize the template by creating your own component that extends the default template. For example:

System Attribute	Type	Description
		<code>&lt;aura:component extends="aura:template" ... &gt;</code>
useAppcache	Boolean	Specifies whether to use the application cache. Valid options are <code>true</code> or <code>false</code> . Defaults to <code>false</code> .

`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:

[URL-Centric Navigation](#)

[App Basics](#)

[Using the AppCache](#)

[Application Access Control](#)

## aura:component

An Aura component is represented by the `aura:component` tag, which has the following optional attributes.

Attribute	Type	Description
abstract	Boolean	Set to <code>true</code> if the component is abstract, or <code>false</code> otherwise.
access	String	Indicates whether the component can be used outside of a namespace. Possible values are <code>internal</code> (default), <code>public</code> , and <code>global</code> .
controller	String	The server-side controller class for the component. The format is <code>java://&lt;package.class&gt;</code> .
description	String	A description of the component.
extends	Component	The component to be extended, if applicable. For example, <code>extends="ui:input"</code> .
extensible	Boolean	Set to <code>true</code> if the component can be extended, or <code>false</code> otherwise.
implements	String	A comma-separated list of interfaces that the component implements.
model	String	The model class used to initialize data for the component. The format is <code>java://&lt;package.class&gt;</code> .
render	String	Renders the component using client-side or server-side renderers. If not provided, Aura determines any dependencies and whether the component should be rendered client- or server-side.  Valid options are <code>client</code> or <code>server</code> . The default is <code>auto</code> .

Attribute	Type	Description
		Specify this attribute in the top-level component. For example, specify <code>render="client"</code> if you want to inspect the component on the client-side during testing.
support	String	The support level for the component. Valid options are <code>PROTO</code> , <code>DEPRECATED</code> , <code>BETA</code> , or <code>GA</code> .

`aura:component` 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 component. In markup, this is everything in the body of the tag.

### See Also:

[Components](#)

[Component Access Control](#)

[Client-Side Rendering to the DOM](#)

[Dynamically Creating Components](#)

## aura:clientLibrary

The `<aura:clientLibrary>` tag enables you to specify JavaScript or CSS libraries that you want to use. Use the tag in a `.cmp` or `.app` resource.

Here is some example markup for including client libraries in a component.

```
<!-- External URL -->
<aura:clientLibrary url="http://jquery.org/latest/jquery.js" type="JS" />
<!-- Absolute path for local library-->
<aura:clientLibrary url="/absolute/path/to/file.js" type="JS" />
<!-- Relative path for local library-->
<aura:clientLibrary url="relative/path/to/file.css" type="CSS" />
```

The `<aura:clientLibrary>` tag includes these system attributes.

System Attribute	Description
combine	<p>If set to <code>true</code>, the library is added to <code>resources.js</code> or <code>resources.css</code>. This option is only available for resources that are available on the local server, for example under the <code>aura-resources</code> folder.</p> <p>Combining libraries into one file can improve performance by reducing the number of requests, instead of a separate request for each library.</p>
modes	A comma-separated list of modes that use the client library. If no value is set, the library is available for all modes.



System Attribute	Description
name	<p>The name of a <code>ClientLibraryResolver</code> that provides the URL. The name attribute is useful if the location or URL of the library needs to be dynamically generated.</p> <p>The name attribute is required if the <code>url</code> attribute is not specified; otherwise, it's ignored. See <a href="#">Add a Client Library Resolver</a> on page 201.</p>
type	The type of library. Values are CSS, or JS for JavaScript.
url	<p>The external URL or path to the file on the server for the library. Examples are:</p> <pre>http://jquery.org/latest/jquery.js</pre> <pre>/absolute/path/to/file.js</pre> <pre>relative/path/to/file.css</pre>

## Add a Client Library Resolver

1. Create a class that extends the `ClientLibraryServiceImpl` Java class.

```
import org.auraframework.def.ClientLibraryDef;
import org.auraframework.impl.clientlibrary.resolver.AuraResourceResolver;
import org.auraframework.impl.clientlibrary.ClientLibraryServiceImpl;

public class SampleClientLibraryService extends ClientLibraryServiceImpl {
    ...
}
```

2. In the constructor, register your new resolver that points to the client library. For example, to register a MadLib external JavaScript library:

```
public SampleClientLibraryService() {
    super();
    // Register external JavaScript library
    // This is a just a sample. Resolvers are more useful if the URL
    // needs to be dynamically generated.
    getResolverRegistry().register(new AuraResourceResolver(
        "MadLib", ClientLibraryDef.Type.JS,
        "http://www.docsample.org/madlib.js",
        "http://www.docsample.org/madlib.js"));
}
```

3. Create a new configuration class to direct the service loader to use the new `SampleClientLibraryService` class instead of the default `ClientLibraryServiceImpl` class. Note that Spring looks for this class in the configuration package.

```
package configuration;

@AuraConfiguration
public class SampleLibraryServiceConfig {
    @Impl
    @Primary
    public ClientLibraryService customClientLibraryService() {
        return new SampleClientLibraryService();
    }
}
```

```
    }  
}
```

**See Also:**  
[Styling Apps](#)  
[Using JavaScript Libraries](#)

# aura:dependency

The `<aura:dependency>` tag enables you to declare dependencies that can't easily be discovered by Aura.

The Aura framework automatically tracks dependencies between definitions, such as components. This enables the framework to automatically reload when it detects that you've changed a definition during development. However, if a component uses a client- or server-side provider that instantiates components that are not 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 component and its dependencies are sent to the client, when needed.

For example, adding this tag to a component marks the `aura:placeholder` component as a dependency.

```
<aura:dependency resource="markup://aura:placeholder" />
```

The `<aura:dependency>` tag includes these system attributes.

System Attribute	Description
resource	<p>The resource that the component depends on. For example, <code>resource="markup://sampleNamespace:sampleComponent"</code> refers to the <code>sampleComponent</code> in the <code>sampleNamespace</code> namespace.</p> <p>Use an asterisk (*) in the resource name for wildcard matching. For example, <code>resource="markup://sampleNamespace:*" </code> matches everything in the namespace; <code>resource="markup://sampleNamespace:input*" </code> matches everything in the namespace that starts with <code>input</code>.</p>
type	<p>The type of resource that the component depends on. The default value is <code>COMPONENT</code>. Use <code>type="*" </code> to match all types of resources.</p> <p>The most commonly used values are:</p> <ul style="list-style-type: none"><li>COMPONENT</li><li>APPLICATION</li><li>EVENT</li></ul> <p>Use a comma-separated list for multiple types; for example: <code>COMPONENT, APPLICATION</code>.</p>

**See Also:**  
[Client-Side Runtime Binding of Components](#)  
[Server-Side Runtime Binding of Components](#)  
[Dynamically Creating Components](#)

## aura:event

An event is represented by the `aura:event` tag, which has the following attributes.

Attribute	Type	Description
<code>access</code>	String	Indicates whether the event can be extended or used outside of a namespace. Possible values are <code>internal</code> (default), <code>public</code> , and <code>global</code> .
<code>description</code>	String	A description of the event.
<code>extends</code>	Component	The event to be extended. For example, <code>extends="namespace:myEvent"</code> .
<code>type</code>	String	Required. Possible values are <code>COMPONENT</code> or <code>APPLICATION</code> .
<code>support</code>	String	The support level for the event. Valid options are <code>PROTO</code> , <code>DEPRECATED</code> , <code>BETA</code> , or <code>GA</code> .

### See Also:

[Events](#)

[Event Access Control](#)

## aura:if

`aura:if` renders the content within the tag if the `isTrue` attribute evaluates to true.

Aura evaluates the `isTrue` expression on the server and instantiates components either in its body or `else` attribute.



**Note:** `aura:if` instantiates the components in either its body or the `else` attribute, but not both. `aura:renderIf` instantiates both the components in its body and the `else` attribute, but only renders one. If the state of `isTrue` changes, `aura:if` has to first instantiate the components for the other state and then render them. We recommend using `aura:if` instead of `aura:renderIf` to improve performance. Only consider using `aura:renderIf` if you expect to show the components for both the `true` and `false` states, and it would require a server round trip to instantiate the components that aren't initially rendered. Otherwise, use `aura:if` to render content if a provided expression evaluates to true.

Attribute Name	Type	Description
<code>else</code>	ComponentDefRef[]	The markup to render when <code>isTrue</code> evaluates to false. Set this attribute using the <code>aura:set</code> tag.
<code>isTrue</code>	string	Required. An expression that determines whether the content is displayed. If it evaluates to <code>true</code> , the content is displayed.

### Example

The following components in the `auradocs` namespace show how to use `aura:if` to conditionally render markup based on the result of evaluating an expression.

sampleIf.cmp extends sampleIfDemo.cmp.

sampleIf.cmp

```
<aura:component extends="auradocs:sampleIfDemo">

<aura:set attribute="show">
  <p>So this component from sampleIfDemo is rendered.</p>
</aura:set>

</aura:component>
```

sampleIfDemo.cmp

```
<aura:component extensible="true">
  <aura:attribute name="show" type="String"/>

  <aura:if.isTrue="{!v.show != null}">
    <p>isTrue evaluates to true: {!v.show}</p>

    <aura:set attribute="else">
      <p>isTrue evaluates to false</p>
    </aura:set>

  </aura:if>
</aura:component>
```

In this example, the body displays when the show attribute contains a value. The show attribute is set in sampleIf.cmp, which extends sampleIfDemo.cmp.

See Also:

[aura:renderIf](#)

aura:interface

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 a namespace. Possible values are internal (default), public, and global.
description	String	A description of the interface.
extends	Component	The comma-seperated list of interfaces to be extended. For example, extends="namespace:intfB".
provider	String	The provider for the interface.
support	String	The support level for the interface. Valid options are PROTO, DEPRECATED, BETA, or GA.

See Also:

[Interfaces](#)

[Interface Access Control](#)

## aura:iteration

`aura:iteration` iterates over a collection of items and renders the body of the tag for each item.

Data changes in the collection are rerendered automatically on the page. `aura:iteration` supports iterations containing components that have server-side dependencies or that can be created exclusively on the client-side.



**Note:** `aura:iteration` replaces `aura:forEach`, which is deprecated.

Attribute Name	Type	Description
<code>body</code>	<code>ComponentDefRef[]</code>	Required. Template to use when creating components for each iteration. You can put any markup in the <code>body</code> . A <code>ComponentDefRef[]</code> stores the metadata of the component instances to create on each iteration, and each instance is then stored in <code>realbody</code> .
<code>end</code>	<code>Integer</code>	The index of the collection to stop at (exclusive).
<code>forceServer</code>	<code>Boolean</code>	Force a server request for the component body. Set to <code>true</code> if the iteration requires any server-side creation. The default is <code>false</code> .
<code>indexVar</code>	<code>String</code>	The variable name to use for the index of each item inside the iteration.
<code>items</code>	<code>List</code>	Required. The collection of data to iterate over.
<code>realbody</code>	<code>Component[]</code>	Do not use. Any value set is ignored. Placeholder for body rendering.
<code>start</code>	<code>Integer</code>	The index of the collection to start at (inclusive).
<code>var</code>	<code>String</code>	Required. The variable name to use for each item inside the iteration.

This example shows how you can use `aura:iteration` exclusively on the client-side.

```
<aura:component>
  <aura:iteration items="1,2,3,4,5" var="item">
    <meter value="{!item / 5}" /><br/>
  </aura:iteration>
</aura:component>
```

The output shows five meters with ascending values of one to five.

### Example Using Data from a Model

This example shows a dynamic iteration involving user input and view update using the controller file.

#### Component source

```
<aura:component render="client" model="java://org.auraframework.docs.SampleIterationModel">
  <aura:attribute name="tochange" type="integer" />
  <aura:attribute name="newvalue" type="string" />
```

```

    <div class="container" aura:id="container">
      <aura:iteration aura:id="iteration" items="{!m.data}" var="stuff" indexVar="index"
start="1" end="6">
        <div aura:id="simple">#{!index}: {!stuff.letters}</div>
      </aura:iteration>
    </div>

    <div>
      Change item #<ui:inputNumber value="{!v.tochange}"/> to <ui:inputText
value="{!v.newvalue}"/>
      <ui:button press="{!c.changeOneValue}" label="Go"/>
    </div>

  </aura:component>

```

### Client-side controller source

```

({
  changeOneValue: function(cmp, evt) {
    var data = cmp.find("iteration").getAttributes().getValue("items");
    var val = data.getValue(cmp.get("v.tochange")).getValue("letters");
    val.setValue(cmp.get("v.newvalue"));
  }
})

```

The sample component iterates over a set of values retrieved from the model. You can update one of the entries and see the change rendered on the page.

In the container `div` element, the `{!m.data}` expression returns `getData()` from the model class. The component iterates through the `List of HashMap` objects returned by `getData()` in the model.

Each item corresponds to `stuff`, denoted by the `var` attribute. So `{!stuff.letters}` in the output displays the value associated with the `letters` key in each `Map`.

When you click **Go** in the component, it calls `changeOneValue` in the client-side controller. The function changes the `tochange` attribute to `newvalue` and the updated value is rerendered on the page.

### Using Models and Providers

The example uses a server-side model, `java://org.auraframework.docs.SampleIterationModel`, which initializes a `HashMap` with the `letters` key. Note that you can't retrieve more data from the server after initial rendering. For example, you can't get more data from the model to support pagination through a data set.

If you are creating a component on the client with a server-side dependency and want to use a provider, use both a client-side provider and a server-side provider. You can use a client-side provider on its own for components without a server-side dependency.

### See Also:

[Client-Side Runtime Binding of Components](#)

[Server-Side Runtime Binding of Components](#)

## aura:renderIf

`aura:renderIf` renders the content within the tag if the `isTrue` attribute evaluates to `true`.

Only consider using `aura:renderIf` if you expect to show the components for both the `true` and `false` states, and it would require a server round trip to instantiate the components that aren't initially rendered. Otherwise, use `aura:if` to render content if a provided expression evaluates to true.

Attribute Name	Type	Description
<code>else</code>	<code>Component[]</code>	The markup to render when <code>isTrue</code> evaluates to <code>false</code> . Set this attribute using the <code>aura:set</code> tag.
<code>isTrue</code>	<code>String</code>	Required. An expression that determines whether the content is displayed. If it evaluates to <code>true</code> , the content is displayed.

## Example

The following components show a basic way to use `aura:renderIf` to conditionally render markup based on the result of evaluating an expression.

`sampleRender.cmp` extends `sampleRenderMe.cmp`.

### sampleRender.cmp

```
<aura:component extends="auradocs:sampleRenderMe">
    <aura:set attribute="desc">
        <p>Content from sampleRender</p>
    </aura:set>
</aura:component>
```

### sampleRenderMe.cmp

```
<aura:component extensible="true">
    <aura:attribute name="desc" type="String"/>

    <aura:renderIf isTrue="{!v.desc != null}">
        <p>Evaluate isTrue and render if true: {!v.desc}</p>

        <aura:set attribute="else">
            <p>render content here if isTrue evaluates to false</p>
        </aura:set>
    </aura:renderIf>
</aura:component>
```

In this example, the body displays only if the `desc` attribute contains a value. The `desc` attribute is set in `sampleRender.cmp`, which extends `sampleRenderMe.cmp`.

## Passing in an Expression

Use `aura:renderIf` if you are passing an expression into a component to be evaluated. For example, you have a container component that references a component, which has a `aura:renderIf` tag.

**container.cmp**

```
<aura:attribute name="native" type="Boolean" default="true"/>
<auradocs:myCmp value="0.5" native={!v.native || v.native}/>
<ui:button label="Toggle" press="{!c.toggleMe}"/>
```

**myCmp.cmp**

```
<aura:attribute name="native" type="Boolean" default="true"/>
<aura:attribute name="value" type="Decimal"/>

<aura:renderIf isTrue="{!v.native}">
  <meter value="{!v.value}">{!(v.value * 100) + '%'}</meter>
  <aura:set attribute="else">
    <!--your Aura meter here-->
  </aura:set>
</aura:renderIf>
```

The container component has a button which toggles the `native` attribute value.

**containerController.js**

```
((
  toggleMe: function(cmp) {
    cmp.set('v.native', !cmp.get('v.native'));
  }
}))
```

When the button is pressed, the expression `native={!v.native || v.native}` is passed into the `aura:renderIf` tag and reevaluated correctly.

**See Also:**

[aura:if](#)

**aura:set**

Use the `<aura:set>` system tag to set the value of an attribute in a super component, on a component reference, or on an event or interface. When you include another component, such as `<ui:button>`, in a component, we call that a component reference to `<ui:button>`.

To learn more, see:

- [Setting Attributes on a Super Component](#)
- [Setting Attributes on a Component Reference](#)
- [Setting Attributes Inherited from an Interface](#)

**Setting Attributes on a Super Component**

Use the `<aura:set>` system tag to set the value of an attribute in a super component if you are extending a component or implementing an interface.

Every component inherits the `body` attribute from `<aura:component>` so `body` has some special behavior with `<aura:set>`.

As well as the `body` attribute, you can use `<aura:set>` with other inherited attributes.



Let's look at an example. Here is the `auradocs:sampleSetTagBase` component.

### Component source

```
<aura:component extensible="true">
  <aura:attribute name="address1" type="String" />

  sampleSetTagBase address1: {!v.address1}<br/>

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

The `address1` and `body` attributes don't return any values yet as they haven't been set.

Here is the `auradocs:sampleSetTagEx` component that extends `auradocs:sampleSetTagBase`.

### Component source

```
<aura:component extends="auradocs:sampleSetTagBase">
  <aura:set attribute="address1" value="808 State St" />

  sampleSetTagEx address1: {!v.address1}
</aura:component>
```

`sampleSetTagEx` sets a value for the `address1` attribute in the super component, `sampleSetTagBase`. Note that the `{!v.address1}` expression in `auradocs:sampleSetTagEx` outputs an empty value as `aura:set` set the value in `sampleSetTagBase`. The `address1` attribute has a different value at each level of inheritance.



**Warning:** This usage of `<aura:set>` works for components and abstract components, but it doesn't work for interfaces. See [Setting Attributes Inherited from an Interface](#) on page 210 for more information.

If you are using a component by making a reference to it in your component, you can set the attribute value directly in the markup. For example, here is a component that makes a reference to `sampleSetTagEx` and sets the `address1` attribute directly without using `aura:set`.

### Component source

```
<aura:component>
  <auradocs:sampleSetTagEx address1="1 Sesame St" />
</aura:component>
```

This component renders the following output:

```
sampleSetTagBase address1: 808 State St
sampleSetTagBase body: sampleSetTagEx 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 `<ui:button>`, in a component, we call that a component reference to `<ui:button>`. You can use `<aura:set>` to set an attribute on the component reference. For example, if your component includes a reference to `<ui:button>`:

```
<ui:button label="">
  <aura:set attribute="label" value="hello"/>
</ui:button>
```

This is equivalent to:

```
<ui:button label="hello"/>
```

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. In the [Aura Note sample app](#), the `<aura:set>` tag specifies the markup for the `left` attribute in the `ui:block` component.

```
<ui:block class="wrapper" aura:id="block">
  <aura:set attribute="left">
    <auranote:sidebar aura:id="sidebar" />
  </aura:set>
  <auranote:details aura:id="details" />
</ui:block>
```

### See Also:

[Setting Attributes on 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. For example, a component implements an interface that has an attribute `myBoolean` set to `false`. The following example sets `myBoolean` on the component to `true`.

```
<aura:component implements="auradocs:myIntf">
  <aura:attribute name="myBoolean" type="Boolean" default="true" />
</aura:component>
```

If the component that implements the interface is contained in another component, you can use `aura:set` as discussed. Alternatively, you can set the attribute value like this.

```
<aura:component>
  <auradocs:sampleCmp myBoolean="true" />
</aura:component>
```

## Supported HTML Tags

An HTML tag is treated as a first-class component in Aura. Each HTML tag is translated into an Aura component, allowing it to enjoy the same rights and privileges as any other component.

We recommend that you use Aura components in preference to HTML tags. For example, use `ui:button` instead of `<button>`. Aura 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 components that come out-of-the-box with Aura 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>`.

The majority of HTML5 tags are supported.

Aura disallows a few HTML tags deemed to be unsafe or unnecessary. Aura doesn't support these tags:

- `applet`
- `base`
- `basefont`
- `embed`
- `font`
- `frame`
- `frameset`
- `isindex`
- `noframes`
- `noscript`
- `object`
- `param`
- `svg`

### See Also:

[Supporting Accessibility](#)

## Supported aura:attribute Types

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

Attribute Name	Type	Description
<code>access</code>	String	Indicates whether the attribute can be used outside of a namespace. Possible values are <code>internal</code> (default), <code>private</code> , <code>public</code> , and <code>global</code> .
<code>name</code>	String	Required. The name of the attribute. For example, if you set <code>&lt;aura:attribute name="isTrue" type="Boolean" /&gt;</code> on a component called <code>aura:newCmp</code> , you can set this attribute when you instantiate the component; for example, <code>&lt;aura:newCmp isTrue="false" /&gt;</code> .
<code>type</code>	String	Required. The type of the attribute. For a list of basic types supported, see <a href="#">Basic Types</a> .
<code>default</code>	String	The default value for the attribute, which can be overwritten as needed. You can't use an expression to set the default value of an attribute. Instead,

Attribute Name	Type	Description
		to set a dynamic default, use an <code>init</code> event. See <a href="#">Invoking Actions on Component Initialization</a> .
<code>required</code>	Boolean	Determines if the attribute is required. The default is <code>false</code> .
<code>description</code>	String	A summary of the attribute and its usage.
<code>serializeTo</code>	String	<p>For optimization. Determines if the attribute is transported from server to client or from client to server. Attributes are transported in JSON format. Valid values are <code>SERVER</code>, <code>BOTH</code>, or <code>NONE</code>. The default is <code>BOTH</code>.</p> <p>Specify <code>SERVER</code> if you don't want to serialize the attribute to the client.</p> <p>Specify <code>NONE</code> if you don't need the attribute to be serialized at all. For example, use <code>NONE</code> if it's a client-side only attribute. If you have a JavaScript object array that must be accessible to markup but don't have a requirement on how the objects are constructed, you can use <code>&lt;aura:attribute name="myObj" type="List" serializeTo="NONE"&gt;</code>.</p>

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

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



**Note:** All type values are case insensitive except for references to Java classes. In general, everything in Aura markup is case insensitive except for references to JavaScript, CSS, or Java.

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

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

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

```
<aura:attribute name="favoriteColors" type="String[]" />
```

## Object Types

An attribute can have a type corresponding to an Object.

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

For example, you may want to create an attribute of type Object to pass a JavaScript array as an event parameter. In the component event, declare the event parameter using `aura:attribute`.

```
<aura:event type="COMPONENT">
  <aura:attribute name="arrayAsObject" type="Object" />
</aura:event>
```

In JavaScript code, you can set the attribute of type Object.

```
// Set the event parameters
var event = component.getEvent(eventType);
event.setParams({
  arrayAsObject:["file1", "file2", "file3"]
});
event.fire();
```

## Collection Types

Here are the supported collection type values.

type	Example	Description
List	<code>&lt;aura:attribute name="colorPalette" type="List" default="red,green,blue" /&gt;</code>	An ordered collection of items.
Map	<code>&lt;aura:attribute name="sectionLabels" type="Map" default="{ a: 'label1', b: 'label2' }" /&gt;</code>	A collection that maps keys to values. A map can't contain duplicate keys. Each key can map to at most one value. Defaults to an empty object, <code>{}</code> . Retrieve values by using <code>component.get("v.sectionLabels")['a']</code> .
Set	<code>&lt;aura:attribute name="collection" type="Set" default="1,2,3" /&gt;</code>	A collection that contains no duplicate elements. The order for set items is not guaranteed. For example, "1,2,3" might be returned as "3,2,1".

### 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"/>
<ui:button press="{!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);
    }
})
```

To retrieve list data from a model, you can use `aura:iteration`. This example retrieves data from a model, assuming that you have set the model attribute on the `aura:component` tag.

```
<aura:attribute name="sizes" type="List"/>
<aura:iteration items="{!m.sizes}" var="size">
    {!size.value}
</aura:iteration>
```

```
/** Server-side Model */
@Model
public class MyModel {
```

```
public List<MyDataType> getSizes() {
    ArrayList<MyDataType> s = new ArrayList<MyDataType>(2);
    //Set list items here
    return s;
}
```

### Setting 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';
```

The following example retrieves data from a map.

```
for (key in myMap) {
    //do something
}
```

### See Also:

[Java Models](#)

[Custom Java Class Types](#)

## Custom Java Class Types

An attribute can have a type corresponding to a Java class. For example, this is an attribute for a `Color` Java class:

```
<aura:attribute name="color" type="java://org.docsample.Color" />
```

If you create a custom Java type, it must implement `JsonSerializable` to enable marshalling from the server to the client. For example, see [Note.java](#) in the Aura Note sample app.

### Support for Collections

If an `<aura:attribute>` can contain more than one element, use a `List` instead of an array.



**Note:** You can't declare an `<aura:attribute>` to be an array of a custom Java type.

The following `aura:attribute` shows the syntax for a `List` of Java objects:

```
<aura:attribute name="colorPalette" type="List" />
```

You can also use `type="java://List"` instead of `type="List"`. Both definitions are functionally equivalent.

```
<aura:attribute name="colorPalette" type="java://List" />
```

## Aura-Specific Types

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

type	Example	Description
<code>Aura.Component</code>	N/A	A single Aura component. We recommend using <code>Aura.Component[]</code> instead.
<code>Aura.Component[]</code>	<pre>&lt;aura:attribute name="detail" type="Aura.Component[]" /&gt;</pre> <p>To set a default value for <code>type="Aura.Component[]"</code>, put the default markup in the body of <code>aura:attribute</code>. For example:</p> <pre>&lt;aura:component&gt;   &lt;aura:attribute     name="detail"     type="Aura.Component[]"&gt;     &lt;p&gt;default paragraph&lt;/p&gt;   &lt;/aura:attribute&gt;   Default value is:   {!v.detail} &lt;/aura:component&gt;</pre>	Use this type to set blocks of markup. An attribute of type <code>Aura.Component[]</code> is called a facet.
<code>Aura.Action</code>	<pre>&lt;aura:attribute name="onclick" type="Aura.Action" /&gt;</pre>	Use this type to pass an Aura action to a component.

**See Also:**

[Component Body](#)  
[Component Facets](#)

**Using the Action Type**

An `Aura.Action` is a reference to an action in Aura. You can pass an `Aura.Action` around so the receiving component can execute the action in its client-side controller.

Use `$A.enqueueAction()` to add client-side or server-side controller actions to the queue of actions to be executed.

The Aura Note sample app uses `Aura.Action` in the `listRow` component.

**listRow.cmp**

```
<aura:component extensible="true">
  ...
  <aura:attribute name="onclick" type="Aura.Action" />
  ...
  <li onclick="{!v.onclick}">
    ...
  </li>
</aura:component>
```

The `onclick` attribute has `type="Aura.Action"`.

**noteListRow.cmp**

```
<aura:component extends="auranote:listRow">
  ...
  <aura:set attribute="onclick" value="{!c.openNote}" />
```



```
...  
</aura:component>
```

The `noteListRow` component extends the `listRow` component and sets the value for the `onclick` attribute in `listRow` to `{!c.openNote}`, which is a reference to an action in the client-side controller for `noteListRow.cmp`. The action is executed when a user clicks the bullet associated with `<li onclick="{!v.onclick}">` in `listRow`.

**See Also:**

[\*Handling Events with Client-Side Controllers\*](#)

## Chapter 28

### Aura Request Lifecycle

---

#### In this chapter ...

- [Initial Application Request](#)
- [Component Request Lifecycle](#)

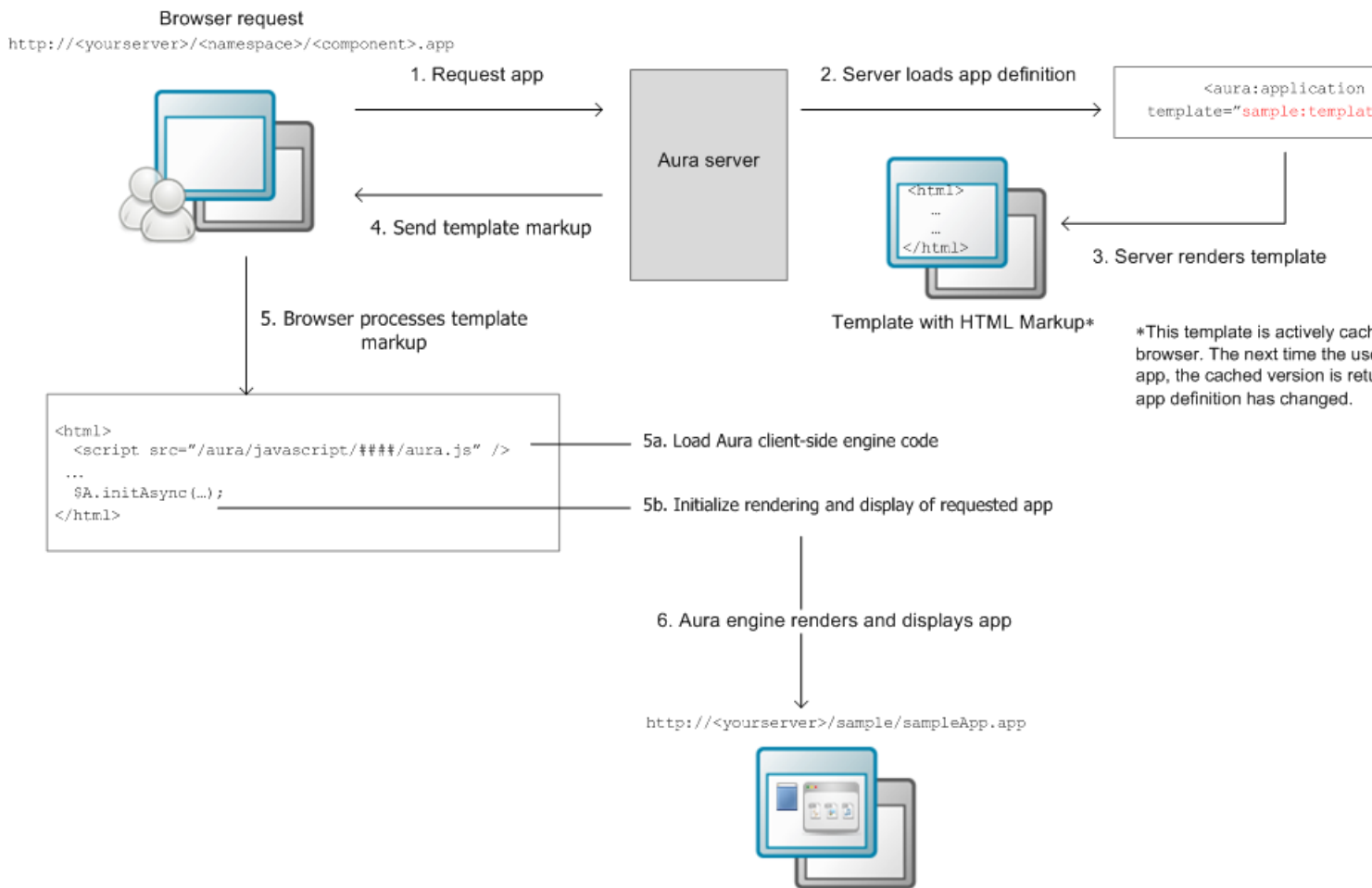
This section shows how Aura handles the initial request for an application, as well as a component request. You can use Aura without knowing these details but read on if you are curious about how things work under the covers.

## Initial Application Request

When you make a request to load an application on a browser, Aura returns an HTTP response with a default template, denoted by the template attribute in the .app file. The template contains JavaScript tags that make requests to get your application data.

The browser renders the specified template and loads the Aura engine and the component definitions in the dependency tree of the app. The Aura engine renders the requested application. The Aura engine processes the application markup, and translates the component markup to HTML objects, returning the DOM elements that are rendered to the browser.

This diagram illustrates the component request lifecycle.



### See Also:

[Component Request Overview](#)

## Component Request Lifecycle

When a component is requested, Aura retrieves the relevant metadata and data from the server to construct the component. The framework uses the metadata and data to construct the component on the client, enabling the client to render the component.

### [Component Request Overview](#)

### [Server-Side Processing for Component Requests](#)

### [Client-Side Processing for Component Requests](#)

### [Component Request Glossary](#)

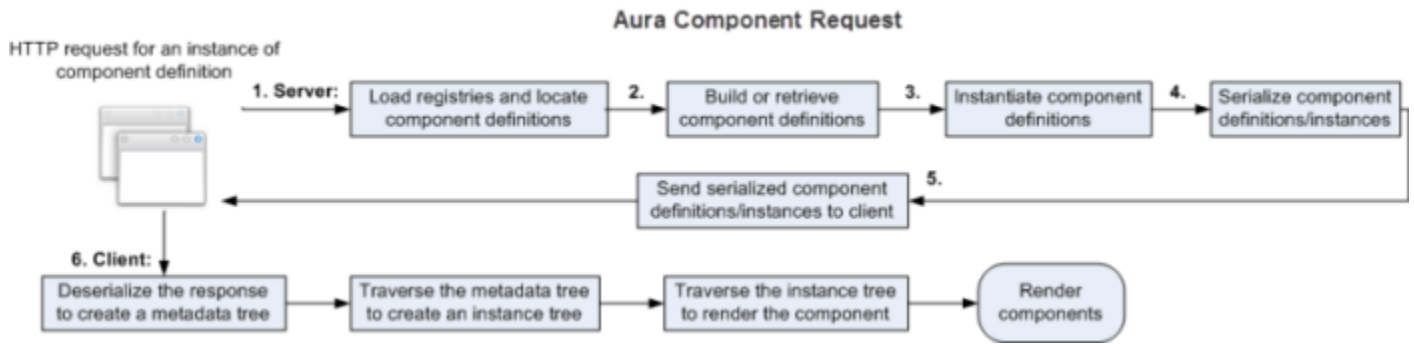
## Component Request Overview

Aura performs initial construction of a component on the server. The client completes the initialization process and manages any rendering or rerendering.

Before we explore the component request process, it's important to understand these terms.

Term	Description
Definition	<p>Each definition describes metadata for an element, such as a component, event, controller, or model. A large part of Aura is a registry of definitions for its various elements.</p> <p>A definition's metadata can include a name, location of origin, and descriptor (DefDescriptor, the primary key of the definition).</p>
DefDescriptor	<p>A DefDescriptor acts as a key for a definition in a registry. It's an Aura class that contains the metadata for any definition used in Aura, such as a component, action, or event. In the example of a model, it is a nicely parsed description of <code>model="java://myPackage.MyClass"</code> with methods to retrieve the language, class name, and package name. Rather than passing a more heavyweight definition around in code, Aura usually passes around a DefDescriptor instead.</p> <p>The qualified name for a DefDescriptor has a format of either <code>prefix://namespace:name</code> or <code>prefix://namespace.name</code>. For example, <code>js://ui.button</code>.</p> <ul style="list-style-type: none"> <li>• <code>prefix</code>: Defines the language, such as JavaScript or Java</li> <li>• <code>namespace</code>: Corresponds to the package name or XML namespace</li> <li>• <code>name</code>: Corresponds to the class name or local name</li> </ul>
Instance	<p>An instance represents the data for a component, event, or action. The component data is contained in its model and attributes.</p>
Registry	<p>Registries store metadata definitions. Some registries last for the duration of a request, while others are cached for the lifetime of the app server. They may be created during the request process and destroyed when the server completes the request. A master definition registry contains a list of registries for each Aura resource.</p>

Let's see what happens when a client requests a component at the server via an HTTP request in the form `http://<yourServer>/namespace/<component>.cmp`.



Here's how a component request is processed on the server and client:

The server:

1. Loads registries and locates component definitions
2. Builds or retrieves component definitions
3. Instantiates component definitions
4. Serializes component definitions and instances
5. Sends serialized component definitions and instances to the client

The client:

1. Deserializes the response to create a metadata tree
2. Traverses the metadata tree to create an instance tree
3. Traverses the instance tree to render the component
4. Renders the component

## See Also:

[Server-Side Processing for Component Requests](#)

[Client-Side Processing for Component Requests](#)

## Server-Side Processing for Component Requests

A component lifecycle starts when the client sends an HTTP request to the server, which can be in the form `http://<yourServer>/<namespace>/<component>.cmp`. Attributes can be included in the query string, such as `http://<yourServer>/<namespace>/<component>.cmp?title=Component1`. If attributes are not specified, the defaults that are defined in the attribute definition are used.

For a component request, the server:

1. Load registries and locates component definitions.
2. Build or retrieves component definitions.
3. Instantiate component definitions.
4. Serialize component definitions and instances.
5. Send serialized component definitions and instances to the client.

### 1. Load registries and locate component definitions.

When the server receives an HTTP request, the Aura framework is loaded according to the specified mode. `AuraContextFilter` creates a `AuraContext`, which contains the mode denoted by the `aura.mode` parameter in the URL, such as in `http://<yourServer>/namespace/<component>.cmp?aura.mode=PROD`. Aura uses the default mode if the `aura.mode` parameter is not included in the query string.

The server receives and parses the request for an instance of a component definition (`ComponentDef`). If attributes are included, Aura converts them to strongly typed attributes for the component definition.

Next, the registries are loaded. Registries store metadata for Aura objects. They may be created during the request process and destroyed when the server completes the request.

A master definition registry (`MasterDefRegistry`) contains a list of registries (`DefRegistry`) that are used to load and cache definitions. A separate registry is used for each Aura object, such as actions, or controllers.

## 2. Build or retrieve component definitions.

This stage of the process retrieves the component's metadata, known as the `ComponentDef`.

After the relevant registries are identified, the server determines if the requested `ComponentDef` is already cached.

- If it's cached in a registry or found in other locations, the `ComponentDef` is returned and the component definition tree is updated to include the definition. The `ComponentDef` is cached, including its references to other `ComponentDefs`, attributes, events, controller, and resources, such as CSS styles.
- If the `ComponentDef` is not cached, the server locates and parses the source code to construct the `ComponentDef`. The server also identifies the language and definition type of the `ComponentDef`.

Any dependencies on other definitions are also determined. Dependencies may include definitions for interfaces, controllers, actions, and models. A `DefRegistry` that doesn't contain the `ComponentDef` passes the request to a `DefFactory`, which builds the definition.

Each component definition in the tree is parsed iteratively. The process is completed when the `ComponentDef` tree doesn't contain any unparsed `ComponentDefs`.

## 3. Instantiate component definitions.

Once the server completes the component definition process, it can create a component instance. To start this instantiation, the `ComponentDef` (a root definition) is retrieved along with any attribute definitions and references to other components. The next steps are:

- **Determine component definition type:** Aura determines whether the root component definition is abstract or concrete.
- **Create component instances:**
  - ◇ **Abstract:** Aura can instantiate abstract component definitions using a provider to determine the concrete component to use at runtime.
  - ◇ **Concrete:** Aura constructs a component instance and any properties associated with it, along with its super component. Attribute values of the component definitions are loaded, and can consist of other component definitions, which are instantiated recursively.
- **Create model instances:** After the super component definition is instantiated, Aura creates any associated component model that hasn't been instantiated.
- **Create attribute instances:** Aura instantiates all remaining attributes. If the attribute refers to an uninstantiated component definition, the latter is instantiated. Non-component attribute values may come from a client request as a literal or expression, which can be derived from a super component definition, a model, or other component definitions. Expressions can be resolved on the client side to allow data to be refreshed dynamically.

The instantiation process terminates when the component and all its child nodes have been instantiated. Note that controllers are not instantiated since they are static and don't have any state.

## 4. Serialize component definition and instances.

Aura enables dynamic rendering on the client side through a JSON serialization process, which begins after instantiation completes. Aura serializes:

- The component instance tree
- Data for the component instance tree

- Metadata for the component instance tree

When the current object has been serialized but it's not the root object corresponding to the requested component, its parent objects are serialized recursively.

## 5. Send serialized component definitions and instances to client.

The server sends the serialized component definitions and instances to the client. Definitions are cached but the instance data is not cached.

The definitions are transmitted in the following format:

```
{ "descriptor": "markup://aura:component",
  "rendererDef": { "serRefId": 2 },
  "attributeDefs": [ { "serId": 20,
    "value": { "descriptor": "body",
      "typeDefDescriptor": "aura://Aura.Component[]",
      "required": false } },
    "interfaces": [ "markup://aura:rootComponent" ],
    "isAbstract": true }
```

The component instance tree is transmitted in the following format:

```
$A.initAsync({ "context": { "mode": "DEV", "app": "auradocs:sample",
  "requestedLocales": [ "en_US", "en" ] },
  "deftype": "APPLICATION",
  "descriptor": "markup://auradocs:sample",
  "host": "",
  "lastmod": 1323498293847 });
```

## See Also:

[Server-Side Runtime Binding of Components](#)

[Initial Application Request](#)

[Component Request Glossary](#)

# Client-Side Processing for Component Requests

After the server processes the request, it returns the component definitions (metadata for the all required components) and instance tree (data) in JSON format.

The client performs these tasks:

1. Deserialize the response to create a metadata tree.
2. Traverse the metadata tree to create an instance tree.
3. Traverse the instance tree to render the component.
4. Render the components.

## 1. Deserialize the response to create a metadata tree.

The JSON representation of the component definition is deserialized to create a metadata structure (JavaScript objects or maps).

## 2. Traverse the metadata tree to create an instance tree.

The client traverses the JavaScript tree to initialize objects from the deserialized tree. The tree can contain:

- Definition: The client initializes the definition.

- Descriptor only: The client knows that definition has been pre-loaded and cached.

### 3. Traverse the instance tree to render the component.

The client traverses the instance tree to render the component instance. The reference IDs are used to recreate the component references, which can point to a `ComponentDef`, a model, or a controller.

### 4. Render the components.

The client locates the renderer definition in the component bundle, or uses the default renderer method to render the component and any sub-components.

#### See Also:

[Server-Side Rendering to the DOM](#)

[Initial Application Request](#)

[Component Request Glossary](#)

## Component Request Glossary

This glossary explains terms related to Aura definitions and registries.

Definition-related Term	Example	Description
Definition	<code>aura:component</code>	<p>Each definition describes metadata for an object, such as a component, event, controller, or model. A large part of Aura is a registry of definitions for its various objects.</p> <p>A definition's metadata can include a name, location of origin, and descriptor (<code>DefDescriptor</code>, the primary key of the definition).</p> <p>A component definition can be used by other component definitions and can extend another component definition.</p>
Root Definition	<code>ComponentDef</code> <code>InterfaceDef</code> <code>EventDef</code>	Top-level definition. Markup language for a root definition can include a pointer to another definition, and references to the descriptors of associate definitions.
Associate Definition	<code>ControllerDef</code> <code>ModelDef</code> <code>ProviderDef</code> <code>RendererDef</code> <code>StyleDef</code> <code>TestSuiteDef</code>	Associate definitions represent objects that are associated with a root definition. An instance of an associate definition can be shared by multiple root definitions. Associate definitions have their own factories, parsers, and caching layers.



Definition-related Term	Example	Description
Subdefinition	AttributeDef RegisterEventDef ActionDef TestCaseDef ValueDef	<p>Subdefinitions can be used to define root definitions or associate definitions. They are stored directly on their parent definitions.</p> <p>For example, a <code>ComponentDef</code> can include multiple <code>AttributeDef</code> objects, and a <code>ControllerDef</code> can include multiple <code>ActionDef</code> objects.</p>
Definition Reference	DefRef ComponentDefRef AttributeDefRef	<p>A subdefinition that points to another definition. At runtime, it can be turned into an instance of the definition to which it points.</p> <p>For example, when a component is instantiated, the component definition can include attribute definition references for each component attribute. The attribute definition reference points to the underlying attribute definition.</p>
Provider		For abstract definition types. A provider determines the concrete <code>ComponentDef</code> to instantiate for each abstract <code>ComponentDef</code> . A provider enables an abstract component definition to be used directly in markup.

Registry-related Terms	Example	Description
Master Definition Registry	MasterDefRegistry	MasterDefRegistry is a top-level DefRegistry that lives for the duration of a request. It is a thin redirector to various long-lived definition registries that load and cache definitions.
Definition Registry	DefRegistry	A DefRegistry loads and caches a list of definitions, such as ActionDef, ApplicationDef, ComponentDef, or ControllerDef. A separate registry is used for all Aura objects. If the definition is not found, the request is passed to DefFactory, an interface that builds the definition.
Definition Descriptor	DefDescriptor	A DefDescriptor acts as a key for a definition in a registry. It's a class that contains the metadata for any definition used in Aura, such as a component, action, or event. In the example of a model, it is a nicely parsed description of <code>model="java://myPackage.MyClass"</code> with methods to retrieve the language, class name, and package name. Rather than passing a more heavyweight definition around in code, Aura usually passes around a DefDescriptor instead.

Registry-related Terms	Example	Description
		<p>The qualified name for a <code>DefDescriptor</code> has the format <code>prefix://namespace:name</code>.</p> <ul style="list-style-type: none"><li>• <code>prefix</code>: Defines the language, such as JavaScript or Java</li><li>• <code>namespace</code>: Corresponds to the package name or XML namespace</li><li>• <code>name</code>: Corresponds to the class name or local name</li></ul>

# Index

\$Browser 30  
 \$Label 30, 61  
 \$Locale 30–31

## A

Access control 160  
 Access control, application 161  
 Access control, attribute 162  
 Access control, component 161  
 Access control, event 162  
 Access control, interface 161  
 Accessibility  
   buttons 69  
   carousels 69  
   events 72  
   help and error messages 69  
   images 70  
   images, informational and decorative 71  
 Actions  
   background 133  
   caboose 134  
   calling server-side 131  
   queueing 132  
   storable 134  
 Adapters  
   overriding 187  
 Anti-patterns  
   events 89  
 Application  
   attributes 198  
   aura:application 198  
   building and running 5  
   creating 94  
   creating, from command line 6  
   creating, in Eclipse 6  
   initial request 219  
   layout and UI 95  
   styling 98  
 Application cache  
   browser support 157  
   enabling 158  
   loading 158  
   overview 157  
   specifying resources 158  
 Application templates  
   external CSS 96  
   JavaScript libraries 96  
 Applications  
   overview 95  
 Apps  
   overview 95  
 Attribute types  
   Aura.Action 215–216

Attribute types (*continued*)  
   Aura.Component 215  
   basic 212  
   collection 214  
   custom Java class 215  
   Object 213  
 Attribute value, setting 208  
 Attributes  
   component reference, setting on 210  
   interface, setting on 210  
   JavaScript 102  
   super component, setting on 208  
 Aura  
   request lifecycle 218  
 Aura Note, sample app 9  
 Aura source  
   building 8  
 aura:application 198  
 aura:attribute 211  
 aura:clientLibrary 200  
 aura:component 199  
 aura:dependency 202  
 aura:event 203  
 aura:if 203  
 aura:interface 204  
 aura:iteration 205  
 aura:renderIf 206  
 aura:set 208, 210  
 aura:template 96  
 Aura.Action 216

## B

Benefits 2  
 Best practices  
   events 88  
 Body  
   JavaScript 102  
 Browser support 3

## C

Client-side controllers 74  
 Component  
   abstract 151  
   adding to an app 7  
   attributes 16  
   aura:component 199  
   aura:interface 204  
   body, setting and accessing 19  
   definition 224  
   documentation 23  
   iteration 205  
   metadata 224

- Component (*continued*)
  - namespace and directory 7
  - nest 17
  - registry 224
  - rendering conditionally 203, 206
  - rendering lifecycle 89
  - request lifecycle 220
  - request overview 220
  - themes, vendor prefixes 98
- Component attributes
  - inheritance 149–150
  - super component 150
- Component body
  - JavaScript 102
- Component bundles 11, 13
- Component definitions
  - dependency 202
- Component facets 20
- Component request
  - client-side processing 223
  - Server-side processing 221
- ComponentDefRef 141
- Components
  - creating 118
  - creating server-side 141
  - HTML markup, using 14
  - ID, local and global 13
  - markup 11
  - modifying 120
  - overview 11
  - styling with CSS 15
  - support level 11
  - unescaping HTML 14
- Controllers
  - calling server-side actions 131
  - client-side 74
  - creating 129
- Converters
  - examples 193
  - registering 192
- Cookbook
  - Java 140
  - JavaScript 115
- CSS
  - external 200
- Custom Converters
  - examples 193
  - registering 192

## D

- Data changes
  - detecting 117
- Debugging
  - mode 167
  - querying state and statistics 184
  - test assertions 166

- Debugging (*continued*)
  - user interactions 167
- DefDescriptor 142
- Demos 9
- Detecting
  - data changes 117
- DOM 101

## E

- Errors 112
- Event handlers 119–120
- Events
  - anti-patterns 89
  - application 79–80
  - aura:event 203
  - best practices 88
  - component 75, 77
  - demo 84
  - example 77, 80
  - firing from non-Aura code 88
  - handling 83
- Examples
  - converters 193
- Exceptions 138
- Expressions
  - examples 28
  - functions 35
  - operators 32
- External CSS 200
- External JavaScript 200

## F

- Fields 110

## G

- globalID 30

## H

- Helpers 103
- HTML, supported tags 211
- HTML, unescaping 14

## I

- Inheritance
  - super component 150
- InstanceService 141
- Integration service 189–190
- Interfaces
  - marker 152
- Introduction 1

## J

Java  
     controllers [129](#)  
 Java cookbook [140](#)  
 JavaScript  
     attribute values [102](#)  
     component [102](#)  
     external [200](#)  
     libraries [101](#)  
     sharing code in bundle [103](#)  
 JavaScript cookbook [115](#)  
 JSON [126](#)

## L

Label  
     setting via parent attribute [66](#)  
 Label parameters [62](#)  
 Labels  
     dynamically creating [64](#)  
 Lazy loading [21](#)  
 Localization [22](#)  
 Log messages [179](#)

## M

Markup [121](#)  
 Mocking  
     Java actions [173](#)  
     Java models [171](#)  
     Java providers [172](#)  
     overview [170](#)  
 Models  
     Java [124](#)  
     JSON [126](#)  
 Modes [174–175](#), [177](#)

## N

Navigation, metadata-driven  
     custom events [146](#)  
 Navigation, url-centric  
     tokenized event attributes [144](#)  
 Navigation,url-centric  
     custom events [144](#)

## O

Object-oriented development  
     inheritance [148](#)  
     super component [150](#)

## P

Providers [108](#), [137](#)

## Q

Queueing  
     queueing server-side actions [132](#)

## R

Reference  
     overview [197](#)  
 Renderers [105](#), [136](#)  
 Rendering lifecycle [89](#)  
 Request  
     application [219](#)  
 Request lifecycle [218](#)

## S

Server-Side Controllers  
     action queueing [132](#)  
     calling actions [131](#)  
 Services [113](#)  
 Source code [8](#)  
 Storable actions [134](#)  
 Storage service  
     adapters [153](#)  
     initializing [155](#)  
     MemoryAdapter [153](#)  
     SmartStore [153](#)  
     using [156](#)  
     WebSQL [153](#)  
 Styles [121](#)

## T

Terminology [124](#)  
 Testing  
     components [163](#)  
     mode [163](#)  
     sample test cases [169](#)  
     test setup [164](#)  
     utility functions [167](#)  
 Themes  
     vendor prefixes [98](#)  
 TodoMVC, sample app [9](#)

## U

ui components  
     aura:component inheritance [38](#)  
     autocomplete [55](#)  
     block [53](#)  
     button [40](#)  
     checkbox [48](#)  
     inputCurrency [43](#)  
     inputDate [42](#)  
     inputDateTime [42](#)  
     inputDefaultError [49](#)

ui components (*continued*)

- [inputEmail](#) [45](#)
- [inputNumber](#) [43](#)
- [inputPercent](#) [43](#)
- [inputPhone](#) [45](#)
- [inputRange](#) [43](#)
- [inputRichText](#) [45](#), [47](#)
- [inputSearch](#) [45](#)
- [inputSelect](#) [50](#)
- [inputText](#) [45](#)
- [inputTextArea](#) [45](#)
- [inputURL](#) [45](#)
- [list](#) [57](#)
- [message](#) [49](#)
- [outputCurrency](#) [43](#)
- [outputDate](#) [42](#)
- [outputDateTime](#) [42](#)
- [outputEmail](#) [45](#)
- [outputNumber](#) [43](#)

ui components (*continued*)

- [outputPercent](#) [43](#)
- [outputPhone](#) [45](#)
- [outputRichText](#) [45](#), [47](#)
- [outputText](#) [45](#)
- [outputTextArea](#) [45](#)
- [outputURL](#) [45](#)
- [vbox](#) [54](#)

[ui components overview](#) [39](#)

**V**

[Validation](#) [110](#)

[Value providers](#)

- [\\$Label](#) [61](#)

[Version numbers](#) [3](#)

**W**

[Warnings](#) [179](#)