Aura Open Source Developer Guide

Last updated: January 4, 2017

CONTENTS

CHAPTER 1 What is Aura?

In this chapter ...

Aura is a UI framework for developing dynamic web apps for mobile and desktop devices. It's a modern framework for building single-page applications engineered for growth.

The framework 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?

The benefits include an out-of-the-box set of components, event-driven architecture, and a framework optimized for performance.

Out-of-the-Box Component Set

Comes with an out-of-the-box set of components 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

Uses a stateful client and stateless server architecture that relies on JavaScript on the client side to manage UI component metadata and application data. The client calls the server only when absolutely necessary; for example to get more metadata or data. The server only sends data that is needed by the user to maximize efficiency. The framework uses JSON to exchange data between the server and the client. It intelligently utilizes your server, browser, devices, and network so you can focus on the logic and interactions of your apps.

Event-driven architecture

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.

Faster development

Empowers teams to work faster with out-of-the-box components that function seamlessly with desktop and mobile devices. 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. The framework enables you to 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 use responsive design and provide an enjoyable user experience. Aura supports the latest in browser technology such as HTML5, CSS3, and touch events.

Components

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

The framework includes a 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:

Creating Components

Events

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

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

There are two types of events in the framework:

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

You write the handlers in JavaScript controller actions.

SEE ALSO:

Communicating with Events

Handling Events with Client-Side Controllers

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 ...

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 1.8
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.

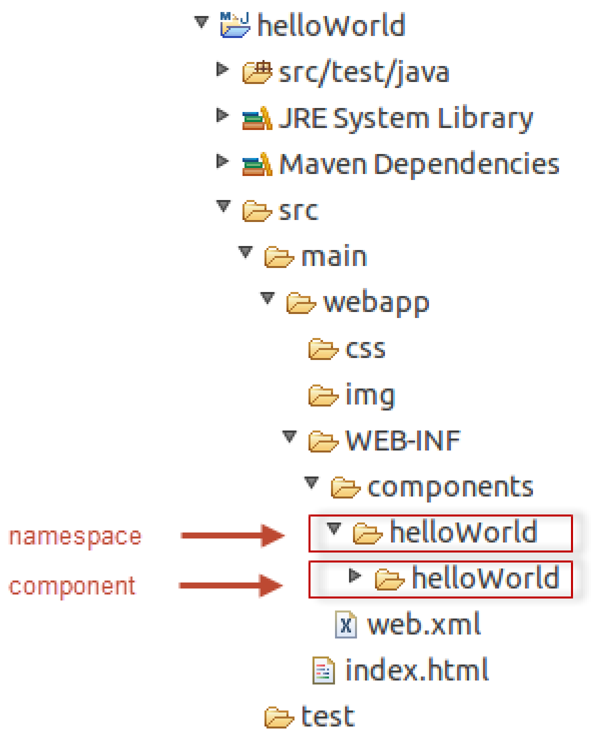
Add a Component

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.

Note: Component and app names must be unique in the namespace and they don't have to match the namespace name.



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
- 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 1.8
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**.

   Note: Running `aura-jetty` enables you to test the doc app and run tests against it. You can't build custom apps using this method.

8. Enter `jetty:run` in the **Goals** field.
9. Click **Apply**.
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.

SEE ALSO:

Reference Doc App

CHAPTER 3 Creating Components

In this chapter ...

Components are the functional units of Aura.

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

Component Markup

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 component definition.

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

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

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

Component Naming Rules

A component name must follow these naming rules:

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

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:

aura:component
Component Access Control
Client-Side Rendering to the DOM
Dynamically Creating Components

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

Note where this component file is stored in the filesystem:
`aura-components/components/docsample/helloWorld/helloWorld.cmp`

All core components are in the `aura-components/components` directory. All folders within that directory map to a 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.

Namespaces in Code Samples

The code samples throughout this guide use the `c` namespace. This namespace has no special significance for open source usage. You can replace the `c` namespace with any other namespace that you prefer to use for development.

Salesforce developers build Lightning components that are based on the open source Aura framework. The `c` namespace is the default namespace if you haven't set a namespace prefix for your Salesforce organization. Using the `c` namespace in our code samples makes it easier for you to reuse the code in both open source Aura components and Lightning components.

Viewing Components

How do we view a component in a Web browser?

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

1. Start the Jetty server on port 8080.

   ```
   mvn jetty:run
   ```

   To use another port, append `-Djetty.port=portNumber`. For example:

   ```
   mvn jetty:run -Djetty.port=9877
   ```

2. Create a component in `aura-components/components/docsample/helloWorld/helloWorld.cmp`. Add this markup to the component.

   ```
   <!--docsample:helloWorld-->
   <aura:component>
       Hello, world!
   </aura:component>
   ```

3. View your component in a browser by navigating to:

   `http://localhost:8080/helloWorld/helloWorld.cmp`

   You should see a simple greeting in your browser.

4. To stop the Jetty server and free up the port when you are finished, press `CTRL+C` on the command line.

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	`sample.cmp` or `sample.app`	The only required resource in a bundle. Contains markup for the component or app. Each bundle contains only one component or app resource.	Creating Components on page 11 aura:application on page 265
CSS Styles	`sample.css`	Contains styles for the component.	CSS in Components on page 16
Controller	`sampleController.js`	Contains client-side controller methods to handle events in the component.	Handling Events with Client-Side Controllers on page 95
Documentation	`sample.auradoc`	A description, sample code, and one or multiple references to example components	Providing Component Documentation on page 50
Model	`sampleModel.js`	JSON model to initialize a component.	JSON Models on page 176
Renderer	`sampleRenderer.js`	Client-side renderer to override default rendering for a component.	Client-Side Rendering to the DOM on page 150
Helper	`sampleHelper.js`	Helper methods that are shared by the controller and renderer.	Sharing JavaScript Code in a Component Bundle on page 147
Provider	`sampleProvider.js`	Client-side provider that returns the concrete component to use at runtime.	Client-Side Runtime Binding of Components on page 153
Test Cases	`sampleTest.js`	Contains a test suite to be run in the browser.	Testing and Debugging Components on page 210

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

Component IDs

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

Local IDs

A local ID is an ID that is only scoped to the component. A local ID enables you to retrieve a component by its ID in JavaScript code. A local ID is often unique but it's not required to be unique.

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

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

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

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

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

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

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

Global IDs

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

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();
```

SEE ALSO:

Finding Components by ID

Which Button Was Pressed?

HTML in Components

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

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

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

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

Warning: Some tags, like `<applet>` and `<font>`, aren't supported. For a full list of unsupported tags, see Supported HTML Tags on page 284.

Unescaping HTML

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

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

`{!`***`expression`***`}` is the framework's expression syntax. For more information, see Using Expressions on page 24.

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`. The framework 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 131.

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. The framework throws an error if a CSS file doesn't follow this convention.

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

Component source

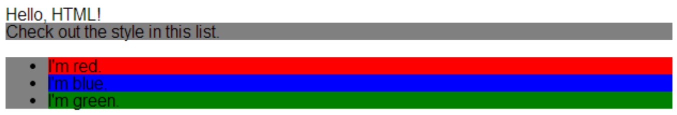
```
<aura:component>
  <div class="white">
    Hello, HTML!
  </div>

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

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

CSS source

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

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

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

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

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

Output



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

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

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

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

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

SEE ALSO:

Adding and Removing Styles

HTML in Components

Component Attributes

Component attributes are like member variables on a class in 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 to add an attribute to the component or app. Let's look at the following sample, `helloAttributes.app`:

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

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

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

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

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.

Attribute Naming Rules

An attribute name must follow these naming rules:

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

Expressions

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

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

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

Attribute Validation

We defined the set of valid attributes in `helloAttributes.app`, so the framework automatically validates that only valid attributes are passed to that component.

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

SEE ALSO:

Supported aura:attribute Types

Using 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. We will first create a few simple components: `c:helloHTML` and `c:helloAttributes`. Then, we'll create a wrapper component, `c:nestedComponents`, that contains the simple components.

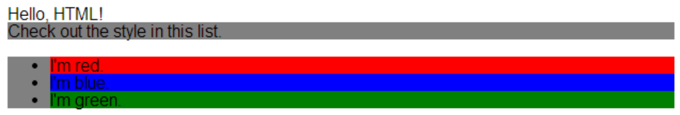
Here is the source for `helloHTML.cmp`.

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

  <h2>Check out the style in this list.</h2>
  <ul>
    <li class="red">I'm red.</li>
    <li class="blue">I'm blue.</li>
    <li class="green">I'm green.</li>
  </ul>
</aura:component>
```

CSS source

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

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

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

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

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

Output



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

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

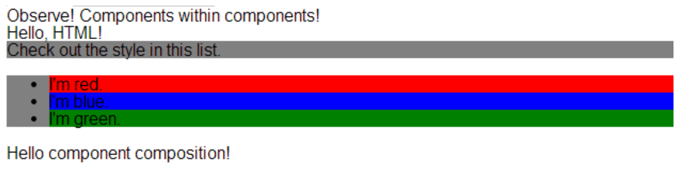
Here is the source for `helloAttributes.cmp`.

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

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

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

    <c:helloHTML/>

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

Output


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

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

Hello component composition!

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

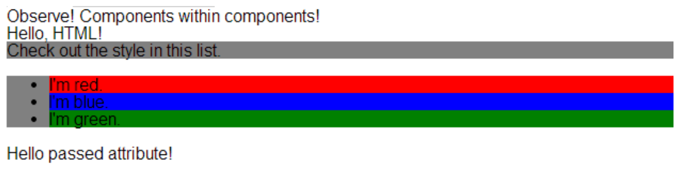
Note how `nestedComponents.cmp` also references `c:helloAttributes`. Just like adding attributes to an HTML tag, you can set attribute values in a component as part of the component tag. `nestedComponents.cmp` sets the `whom` attribute of `helloAttributes.cmp` to "component composition".

Attribute Passing

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

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

    <c:helloHTML/>

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

Output


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

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

Hello passed attribute!

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

Note: `{#v.passthrough}` is an unbound expression. This means that any change to the value of the `whom` attribute in `c:helloAttributes` doesn't propagate back to affect the value of the `passthrough` attribute in `c:nestedComponents2`. For more information, see Data Binding Between Components on page 27.

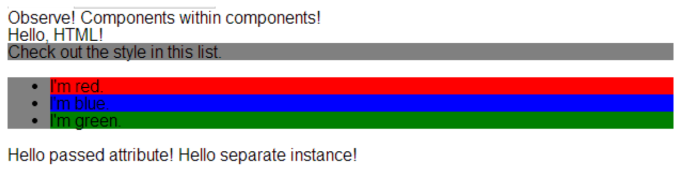
Definitions versus Instances

In object-oriented programming, there's a difference between a class and an instance of that class. Components have a similar concept. When you create a `.cmp` 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 `c:helloAttributes` with a different attribute value. The two instances of the `c:helloAttributes` component have different values for their `whom` attribute .

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

    <c:helloHTML/>

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

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

Output



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

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

Hello passed attribute! Hello separate instance!
```

Component Body

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

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

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

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

Setting the Body Content

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

```
<aura:component>
    <!--START BODY-->
    <div>Body part</div>
    <ui:button label="Push Me"/>
    <!--END BODY-->
</aura:component>
```

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

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

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

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

```
<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.

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

```
<!--c:facetHeader-->
<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. Let's create another component, `helloFacets.cmp`, that sets these attributes.

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

    <c:facetHeader>

        Nice body!

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

</aura:component>
```

Note that `aura:set` sets the value of the `header` attribute of `facetHeader.cmp`, but you don't need to use `aura:set` if you're setting the `body` attribute.

SEE ALSO:

Component Body

Best Practices for Conditional Markup

Use the `<aura:if>` tag to conditionally display markup. Alternatively, you can conditionally set markup in JavaScript logic. Consider the performance cost as well as code maintainability when you design components. The best design choice depends on your use case.

Consider Alternatives to Conditional Markup

Here are some use cases where you should consider alternatives to `<aura:if>`.

You want to toggle visibility

Don't use `<aura:if>` to toggle markup visibility. Use CSS instead. See Dynamically Showing or Hiding Markup on page 170.

You need to nest conditional logic or use conditional logic in an iteration

Using `<aura:if>` can hurt performance by creating a large number of components. Excessive use of conditional logic in markup can also lead to cluttered markup that is harder to maintain.

Consider alternatives:

- Use JavaScript logic in an `init` event handler instead. See Invoking Actions on Component Initialization on page 155.
- Use a provider to determine the concrete component to use at runtime. This can reduce conditional markup and result in cleaner, more maintainable components. See Server-Side Runtime Binding of Components on page 188.

SEE ALSO:

aura:if

Conditional Expressions

Component Versioning

Use versioning to change the behavior of a resource, such as a component, while maintaining backwards compatibility for existing users of the resource. Use versioning in markup in applications, components, interfaces, and events.

You can also use versioning in JavaScript controllers, helpers, and renderers, as well as Java controllers.

Versioning in Component Markup

The `{!Version}` expression returns the version of the component in its current context. The current context is defined by the component using the component.

This component uses `{!Version}` to conditionally change the component behavior depending on its version.

```
<!--c:versionedComponent-->
<aura:component>
    <aura:if isTrue="{!Version > 1.0}">
        <c:newVersion />
    </aura:if>
    <c:originalVersion />
</aura:component>
```

`<cOther:consumingComponent>` uses the `<aura:require>` tag to define the version of `<cOther:versionedComponent>` that it uses. In this example, it requires version 2.0 of any resource in the `c` namespace. This sets the context of `<cOther:versionedComponent>` to version 2.0.

```
<!--cOther:consumingComponent-->
<aura:component>
    <aura:require namespace="c" version="2.0" />
    <c:versionedComponent />
</aura:component>
```

Note: The `{!Version}` expression returns a `String`. Depending on how you define version numbers, you may have to define your own comparison logic to work with numbers or other version identifiers, such as `x.y.z`.

Versioning Quick Reference

This table summarizes how to retrieve the current version in different contexts.

Resource	Returns	Syntax
Markup	`String`	`{!Version}`
JavaScript	`String`	`cmp.getVersion()`
Java	`String`	`Aura.getContextService().getCurrentContext().getAccessVersion()`

Using Expressions

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

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

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

`expression` is a placeholder for the expression.

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

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

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

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

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

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

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

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

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

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

IN THIS SECTION:

Dynamic Output in Expressions

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

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

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

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

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

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

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

SEE ALSO:

Component Attributes

Value Providers

Conditional Expressions

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

Ternary Operator

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

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

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

Using `<aura:if>` for Conditional Markup

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

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

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

SEE ALSO:

Best Practices for Conditional Markup

aura:if

Data Binding Between Components

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

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

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

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

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

Now, let's change the markup from:

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

to:

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

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

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

`{#expression}` (Unbound Expressions)

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

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

`{!expression}` (Bound Expressions)

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

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

Unbound Expressions

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

Here is the markup for `c:childExpr`.

```
<!--c:childExpr-->
<aura:component>
```

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

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

Here is the markup for `c:parentExpr`.

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

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

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

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

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

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

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

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

Navigate to `c:exprApp.app` in your browser.

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

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

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

Navigate to `c:exprApp.app` in your browser again.

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

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

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

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

Navigate to `c:exprApp.app` in your browser again.

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

Warning: Don't use a component's `init` event and client-side controller to initialize an attribute that is used in an unbound expression. The attribute will not be initialized. Use a bound expression instead. For more information on a component's `init` event, see Invoking Actions on Component Initialization on page 155.

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

Bound Expressions

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

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

to:

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

Navigate to `c:exprApp.app` in your browser again.

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

Change Handlers and Data Binding

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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

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

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

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

Navigate to `c:exprApp.app` in your browser again.

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

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

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

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

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

Navigate to `c:exprApp.app` in your browser again.

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

SEE ALSO:

Detecting Data Changes with Change Handlers

Dynamic Output in Expressions

Component Composition

Value Providers

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

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

Value Provider	Description	See Also
`m`	A component's model, which enables the component to initialize its data from a dynamic source, such as a database	Reading Initial Component Data with Models
`v`	A component's attribute set. This value provider enables you to access the value of a component's attribute in the component's markup.	Component Attributes
`c`	A component's controller, which enables you to wire up event handlers and actions for the component	Handling Events with Client-Side Controllers

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

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

Global Value Providers

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

Global Value Provider	Description	See Also
`globalID`	The `globalId` global value provider returns the global ID for a component. Every component has a unique `globalId`, which is the generated runtime-unique ID of the component instance.	Component IDs
`$Browser`	The `$Browser` global value provider returns information about the hardware and operating system of the browser accessing the application.	$Browser
`$Label`	The `$Label` global value provider enables you to access labels stored outside your code.	$Label
`$Locale`	The `$Locale` global value provider returns information about the current user's preferred locale.	$Locale

To add your own custom global value providers, see Adding Custom Global Value Providers.

Accessing Fields and Related Objects

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

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

For example, 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>
```

Note: `{#v.note.createdOn}` is an unbound expression. This means that any change to the `value` attribute in `ui:outputDateTime` doesn't propagate back to affect the value of the `note` attribute in the parent component. For more information, see Data Binding Between Components on page 27.

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

IN THIS SECTION:

Adding Custom Global Value Providers

Add a custom global value provider by implementing the `GlobalValueProviderAdapter` interface.

SEE ALSO:

Dynamic Output in Expressions

`$Browser`

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

Attribute	Description
`formFactor`	Returns a `FormFactor` enum value based on the type of hardware the browser is running on. • DESKTOP for a desktop client • PHONE for a phone including a mobile phone with a browser and a smartphone • TABLET for a tablet client (for which `isTablet` returns `true`)
`isAndroid`	Indicates whether the browser is running on an Android device (`true`) or not (`false`).
`isIOS`	Not available in all implementations. Indicates whether the browser is running on an iOS device (`true`) or not (`false`).
`isIPad`	Not available in all implementations. Indicates whether the browser is running on an iPad (`true`) or not (`false`).
`isIPhone`	Not available in all implementations. Indicates whether the browser is running on an iPhone (`true`) or not (`false`).
`isPhone`	Indicates whether the browser is running on a phone including a mobile phone with a browser and a smartphone (`true`), or not (`false`).
`isTablet`	Indicates whether the browser is running on an iPad or a tablet with Android 2.2 or later (`true`) or not (`false`).
`isWindowsPhone`	Indicates whether the browser is running on a Windows phone (`true`) or not (`false`). Note that this only detects Windows phones and does not detect tablets or other touch-enabled Windows 8 devices.

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

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

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

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

`$Locale`

The `$Locale` global value provider returns information about the browser's locale.

These attributes are based on Java's `Calendar`, `Locale` and `TimeZone` classes.

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

Number and Date Formatting

The framework's number and date formatting are based on Java's `DecimalFormat` and `DateFormat` classes.

Attribute	Description	Sample Value
`currencyformat`	The currency format.	"¤#,##0.00;(¤#,##0.00)" ¤ represents the currency sign, which is replaced by the currency symbol.
`dateFormat`	The date format.	"MMM d, yyyy"
`datetimeFormat`	The date time format.	"MMM d, yyyy h:mm:ss a"
`numberformat`	The number format.	"#,##0.###" # represents a digit, the comma is a placeholder for the grouping separator, and the period is a placeholder for the decimal separator. Zero (0) replaces # to represent trailing zeros.
`percentformat`	The percentage format.	"#,##0%"
`timeFormat`	The time format.	"h:mm:ss a"
`zero`	The character for the zero digit.	“0”

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

Component source

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

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

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

SEE ALSO:

Localization

Adding Custom Global Value Providers

Add a custom global value provider by implementing the `GlobalValueProviderAdapter` interface.

SEE ALSO:

- Value Providers
- Overriding Default Adapters
- Default Adapters

Expression Evaluation

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

Operators are a subset of those available in JavaScript, and evaluation order and precedence are generally the same as JavaScript. Parentheses enable you to ensure a specific evaluation order. What you may find surprising about expressions is how often they are evaluated. The framework notices when things change, and trigger re-rendering of any components that are affected. Dependencies are handled automatically. This is one of the fundamental benefits of the 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. For more information, see Conditional Expressions on page 26.

```
<ui:button aura:id="likeBtn"
     label="{!(v.likeId == null) ? 'Like It' : 'Unlike It'}"
     press="{!(v.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 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.

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

Numeric Literals

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

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

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

String Operators

Expressions based on string operators result in string values.

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

String Literals

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

Literal	Usage	Description
string	`'hello world'`	Literal strings must be enclosed in single quotation marks. Double quotation marks are reserved for enclosing attribute values, and must be escaped in strings.
\<escape>	`'\n'`	Whitespace characters: • `\t` (tab) • `\n` (newline) • `\r` (carriage return) Escaped characters:

Literal	Usage	Description
		• `\"` (literal ") • `\'` (literal ') • `\\` (literal \\)
Unicode	`'\u####'`	A Unicode code point. The # symbols are hexadecimal digits. A Unicode literal requires four digits.
null	`null`	A literal null string. Matches the explicit null value and strings with an undefined value.

Comparison Operators

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

Operator	Alternative	Usage	Description

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

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

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

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

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

Operator	Alternative	Usage	Description

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

Logical Operators

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

Operator	Usage	Description
`&&`	`isEnabled && hasPermission`	Returns `true` if both operands are individually true. You must escape the `&&` operator to `&amp;&amp;` to use it in component markup. Alternatively, you can use the `and()` function and pass it two arguments. For example, `and(isEnabled, hasPermission)`.
`||`	`hasPermission || isRequired`	Returns `true` if either operand is individually true.
`!`	`!isRequired`	Unary operator. Returns `true` if the operand is false. This operator should not be confused with the `!` delimiter used to start an expression in `{!`. You can combine the expression delimiter with this negation operator to return the logical negation of a value, for example, `{!!true}` returns `false`.

Logical Literals

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

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

Conditional Operator

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

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

SEE ALSO:

Expression Functions Reference

Expression Functions Reference

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

Math Functions

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

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

String Functions

Function	Alternative	Usage	Description	Corresponding Operator

`concat`	`add`	`concat('Hello ', 'world')`
		`add('Walk ', 'the dog')`
			Concatenates the two arguments.	+

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

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

Label Functions

Function	Usage	Description

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

Informational Functions

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

Comparison Functions

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

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

Boolean Functions

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

Function	Usage	Description	Corresponding Operator
`and`	`and(isEnabled, hasPermission)`	Returns `true` if both arguments are true.	`&&`
`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)

Using Labels

The framework supports various methods to provide labels in your code using the `$Label` global value provider, which accesses labels stored outside your code.

This section discusses how to use the `$Label` global value provider with these methods:

- The `label` attribute in input components
- The `format()` expression function for dynamically populating placeholder values in labels
- The `aura:label` component for populating placeholder values with components or markup in labels
- The `ui:label` component for visual separation with the label's corresponding input component

IN THIS SECTION:

`$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 49.

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.

SEE ALSO:

Localization

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}" />
```

To output a label and dynamically update it, use the `format()` expression function. For example, if you have `np.labelName` set to `Hello {0}`, the following expression returns `Hello World` if `v.name` is set to `World`.

```
{!format($Label.np.labelName, v.name)}
```

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

Getting Labels in JavaScript

Supporting Accessibility

Java Models

Dynamically Populating Label Parameters

Output and update labels using the `format()` expression function.

You can provide a string with placeholders, which are replaced by the substitution values at runtime.

Add as many parameters as you need. The parameters are numbered and are zero-based. For example, if you have three parameters, they will be named `{0}`, `{1}`, and `{2}`, and they will be substituted in the order they're specified.

Let's look at a custom label, `$Label.mySection.myLabel`, with a value of `Hello {0} and {1}`, where `$Label` is the global value provider that accesses your labels.

This expression dynamically populates the placeholder parameters with the values of the supplied attributes.

```
{!format($Label.mySection.myLabel, v.attribute1, v.attribute2)}
```

The label is automatically refreshed if one of the attribute values changes.

The `format()` expression is more concise and preferred to the equivalent `<aura:label>` markup:

```
<aura:label value="{!$Label.mySection.myLabel}">
    {!v.attribute1}
    {!v.attribute2}
</aura:label>
```

Note: Always use the `$Label` global value provider to reference a label with placeholder parameters. You can't set a string with placeholder parameters as the first argument for `format()`. For example, this syntax doesn't work:

```
{!format('Hello {0}', v.name)}
```

Use this expression instead.

```
{!format($Label.mySection.salutation, v.name)}
```

where `$Label.mySection.salutation` is set to `Hello {0}`.

Populating Parameters with Components or Markup

You must use `<aura:label>` instead of `format()` to populate parameters with component or markup. For example:

```
<aura:label value="{!$Label.message.hello}">
    <ui:button>{!v.name}</ui:button>
</aura:label>
```

This example shows how to include a link in a label by substituting the `{0}` parameter with the embedded `ui:outputURL` 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="{!$Label.MySection.LinkLabel}">
    <aura:label value="{!$Label.MySection.LinkLabel}">
        <ui:outputURL value="http://www.salesforce.com" label="Test Link"/>
    </aura:label>
</ui:button>
```

Note: Setting the `labelDisplay` attribute to `false` hides the label provided by the `label` attribute on the `ui:button` component from view, but makes it available to screen readers.

The next example uses a hard-coded label value rather than a value from the label provider. The `{0}` placeholder is replaced by the `Test Link` label at runtime.

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

Getting Labels in JavaScript

You can retrieve labels in JavaScript code. Your code performs optimally if the labels are statically defined and sent to the client when the component is loaded.

Static Labels

Static labels are defined in one string, such as `"$Label.c.task_mode_today"`. The framework parses static labels in markup or JavaScript code and sends the labels to the client when the component is loaded. A server trip isn't required to resolve the label. Use `$A.get()` to retrieve static labels in JavaScript code. For example:

```
var staticLabel = $A.get("$Label.c.task_mode_today");
```

Dynamic Labels

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

```
// Assume the day variable is dynamically generated
// earlier in the code
// THIS CODE WON'T WORK
var dynamicLabel = $A.get("$Label.c." + day);
```

If the label is already known on the client, `$A.get()` displays the label. If the value is not known, an empty string is displayed in `PROD` mode, or a placeholder value showing the label key is displayed in all other modes.

Since the label, `"$Label.c." + day"`, is dynamically generated, the framework can't parse it and send it to the client when the component is requested. `dynamicLabel` is an empty string, which isn't what you want!

There are a few alternative approaches to using `$A.get()` so that you can work with dynamically generated labels.

If your component uses a known set of dynamically constructed labels, you can avoid a server roundtrip for the labels by adding a reference to the labels in a JavaScript resource. The framework sends these labels to the client when the component is requested. For example, if your component dynamically generates `$Label.c.task_mode_today` and `$Label.c.task_mode_tomorrow` label keys, you can add references to the labels in a comment in a JavaScript file, such as a client-side controller or helper.

```
// hints to ensure labels are preloaded
// $Label.Related_Lists.task_mode_today
// $Label.Related_Lists.task_mode_tomorrow
```

If your code dynamically generates many labels, this approach doesn't scale well.

If you don't want to add comment hints for all the potential labels, the alternative is to use `$A.getReference()`. This approach comes with the added cost of a server trip to retrieve the label value.

This example dynamically constructs the label value by calling `$A.getReference()` and updates a `tempLabelAttr` component attribute with the retrieved label.

```
var labelSubStr = "task_mode_today";
var labelReference = $A.getReference("$Label.c." + labelSubStr);
cmp.set("v.tempLabelAttr", labelReference);
var dynamicLabel = cmp.get("v.tempLabelAttr");
```

`$A.getReference()` returns a reference to the label. This **isn't** a string, and you shouldn't treat it like one. You never get a string label directly back from `$A.getReference()`.

Instead, use the returned reference to set a component's attribute value. Our code does this in `cmp.set("v.tempLabelAttr", labelReference);`.

When the label value is asynchronously returned from the server, the attribute value is automatically updated as it's a reference. The component is rerendered and the label value displays.

Note: Our code sets `dynamicLabel = cmp.get("v.tempLabelAttr")` immediately after getting the reference. This code displays an empty string until the label value is returned from the server. If you don't want that behavior, use a comment hint to ensure that the label is sent to the client without requiring a later server trip.

SEE ALSO:

- Using JavaScript
- Input Component Labels
- Dynamically Populating Label Parameters
- Customizing your Label Implementation
- Modes Reference

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:component>
```

This `inner` component contains a text area component and a `label` attribute that's set by the container component.

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

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 *section*	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 `null` depending on your label system implementation.
String *name*	The label name.
Object *params*	A list of parameter values for substitution on the server. This parameter can be `null` 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

Localization

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

The components retrieve the browser's locale information and display the output components 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.

Note: For more information on supported attributes, see the Reference Doc App.

Component source

```
<aura:component>
    <ui:outputDateTime value="2013-10-07T00:17:08.997Z"  timezone="Europe/Berlin"
langLocale="de"/>
</aura:component>
```

The component renders as `Okt. 7, 2015 2:17:08 AM`.

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

Using the Localization Service

The framework's localization service enables you to manage the localization of date, time, numbers, and currencies. These methods are available in the `AuraLocalizationService` JavaScript API.

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

```
var dateFormat = $A.get("$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);
```

The `AuraLocalizationService` JavaScript API provides methods for working with localization. For example, you can compare two dates to check that one is later than the other.

```
var startDateTime = new Date();
//return the date time at end of the day
var endDateTime = $A.localizationService.endOf(d, 'day');
if( $A.localizationService.isAfter(startDateTime,endDateTime)) {
    //throw an error if startDateTime is after endDateTime
}
```

Note: For more information on the localization service, see the JavaScript API in the Reference Doc App.

SEE ALSO:

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 in a 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">
             <img class="icon bLeft" alt="Find" src="path/to/img">
             <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
`<aura:documentation>`	The top-level definition of the DocDef
`<aura:description>`	Describes the component using extensive HTML markup. To include code samples in the description, use the `<pre>` tag, which renders as a code block. Code entered in the `<pre>` tag must be escaped. For example, escape `<aura:component>` by entering `&lt;aura:component&gt;`.

Tag	Description
`<aura:example>`	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 `<aura:example>` tags. • `name`: The API name of the example • `ref`: The reference to the example component in the format `<namespace:exampleComponent>` • `label`: The label of the title

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
`<aura:component>`	`<aura:component description="Represents a button element">`
`<aura:attribute>`	`<aura:attribute name="langLocale" type="String" description="The language locale used to format date value."/>`
`<aura:event>`	`<aura:event type="COMPONENT" description="Indicates that a keyboard key has been pressed and released"/>`
`<aura:interface>`	`<aura:interface description="A common interface for date components"/>`

Tag	Example
`<aura:registerEvent>`	`<aura:registerEvent name="keydown" type="ui:keydown" description="Indicates that a key is pressed"/>`

SEE ALSO:

Reference

Working with Base Lightning Components

Base Lightning components are the building blocks that make up the modern Lightning Experience, Salesforce1, and Lightning Communities user interfaces.

Base Lightning components incorporate Lightning Design System markup and classes, providing improved performance and accessibility with a minimum footprint.

These base components handle the details of HTML and CSS for you. Each component provides simple attributes that enable variations in style. This means that you typically don't need to use CSS at all. The simplicity of the base Lightning component attributes and their clean and consistent definitions make them easy to use, enabling you to focus on your business logic.

You can find base Lightning components in the `lightning` namespace to complement the existing `ui` namespace components. In instances where there are matching `ui` and `lightning` namespace components, we recommend that you use the `lightning` namespace component. The `lightning` namespace components are optimized for common use cases. Beyond being equipped with the Lightning Design System styling, they handle accessibility, real-time interaction, and enhanced error messages.

In subsequent releases, we intend to provide additional base Lightning components. We expect that in time the `lightning` namespace will have parity with the `ui` namespace and go beyond it.

In addition, the base Lightning components will evolve with the Lightning Design System over time. This ensures that your customizations continue to match Lightning Experience and Salesforce1.

Input Control Components

The following components are interactive, for example, like buttons and tabs.

Type	Key Components	Description	Lightning Design System
Button	`lightning:button`	Represents a button element.	Buttons
Button Icon	`lightning:buttonIcon`	An icon-only HTML button.	Button Icons
Button Group	`lightning:buttonGroup`	Represents a group of buttons.	Button Groups
Button Menu	`lightning:buttonMenu`	A dropdown menu with a list of actions or functions.	Menus
	`lightning:menuItem`	A list item in `lightning:buttonMenu`.	
Select	`lightning:select`	Creates an HTML `select` element.	Select
Tabs	`lightning:tab`	A single tab that is nested in a `lightning:tabset` component.	Tabs

Type	Key Components	Description	Lightning Design System
	`lightning:tabset`	Represents a list of tabs.	

Visual Components

The following components provide informative cues, for example, like icons and loading spinners.

Type	Key Components	Description	Lightning Design System
Avatar	`lightning:avatar`	A visual representation of a person.	
Badge	`lightning:badge`	A label that holds a small amount of information.	Badges
Card	`lightning:card`	Applies a container around a related grouping of information.	Cards
Icon	`lightning:icon`	A visual element that provides context.	Icons
Layout	`lightning:layout`	Responsive grid system for arranging containers on a page.	Grid
	`lightning:layoutItem`	A container within a `lightning:layout` component.	
Spinner	`lightning:spinner`	Displays an animated spinner.	Spinners

Field Components

The following components enable you to enter or display values.

Type	Key Components	Description	Lightning Design System
Input	`lighting:input`	Represents interactive controls that accept user input depending on the type attribute.	Forms
Internationalization	`lighting:formattedDateTime`	Displays formatted date and time.	N/A
	`lighting:formattedNumber`	Displays formatted numbers.	
Rich Text Area	`lighting:inputRichText`	A WYSIWYG editor with a customizable toolbar for entering rich text	Rich Text Editor
Text Area	`lighting:textArea`	A multiline text input.	Textarea

Base Lightning Components Considerations

Learn about the guidelines on using the base Lightning components.

We recommend that you don't depend on the markup of a Lightning component as its internals can change in the future. For example, using `cmp.get("v.body")` and examining the DOM elements can wreak havoc should the component markup change down the road. With LockerService enforced, you won't be able to traverse the DOM for components you don't own. Instead of accessing the DOM tree, you can rely on the component tree and take advantage of value binding with component attributes. For example, you'll go far with `cmp.find("myInput").get("v.name")` instead of `cmp.find("myInput").getElement().name`.

Many of the base Lightning components are still evolving and the following considerations can help you while you're building your apps.

lightning:buttonMenu (Beta)

This component contains menu items that are created only if the button is triggered. You won't be able to reference the menu items during initialization or if the button isn't triggered yet.

lightning:formattedDateTime (Beta)

This component provides fallback behavior in Apple Safari 10 and below. The following formatting options have exceptions when using the fallback behavior in older browsers.

- `era` is not supported.
- `timeZoneName` appends `GMT` for short format, `GMT-h:mm` or `GMT+h:mm` for long format.
- `timeZone` supports `UTC`. If another timezone value is used, `lightning:formattedDateTime` uses the browser timezone.

lightning:formattedNumber (Beta)

This component provides the following fallback behavior in Apple Safari 10 and below.

- If `style` is set to `currency`, providing a `currencyCode` value that's different from the locale displays the currency code instead of the symbol. The following example displays `EUR12.34` in fallback mode and `€12.34` otherwise.

```
<lightning:formattedNumber value="12.34" style="currency"
 currencyCode="EUR"/>
```

- `currencyDisplayAs` supports `symbol` only. The following example displays `$12.34` in fallback mode only if the `currencyCode` matches the user's locale currency and `USD12.34` otherwise.

```
<lightning:formattedNumber value="12.34" style="currency"
 currencyCode="USD" currencyDisplayAs="symbol"/>
```

lightning:input (Beta)

The `file` type is not supported. Also, date pickers are available in the following components but they don't inherit the Lightning Design System styling.

- `<lightning:input type="date" />`
- `<lightning:input type="datetime-local" />`

Fields for percentage and currency input must specify a step increment of 0.01 as required by the native implementation.

```
<lightning:input type="number" name="percentVal" label="Enter a percentage value"
formatter="percent" step="0.01" />
<lightning:input type="number" name="currencyVal" label="Enter a dollar amount"
formatter="currency" step="0.01" />
```

When working with checkboxes, radio buttons, and toggle switches, use `aura:id` to group and traverse the array of components. Grouping them enables you to use `get("v.checked")` to determine which elements are checked or unchecked without reaching into the DOM. You can also use the `name` and `value` attributes to identify each component during the iteration. The following example groups three checkboxes together using `aura:id`.

```
<aura:component>
    <form>
      <fieldset>
```

```
        <legend>Select your favorite color:</legend>
        <lightning:input type="checkbox" label="Red"
            name="color1" value="1" aura:id="colors"/>
        <lightning:input type="checkbox" label="Blue"
            name="color2" value="2" aura:id="colors"/>
        <lightning:input type="checkbox" label="Green"
            name="color3" value="3" aura:id="colors"/>
      </fieldset>
    <lightning:button label="Submit" onclick="{!c.submitForm}"/>
    </form>
</aura:component>
```

In your client-side controller, you can retrieve the array using `cmp.find("colors")` and inspect the `checked` values.

lightning:tab (Beta)

This component creates its body during runtime. You won't be able to reference the component during initialization. Referencing the component using `aura:id` might return unexpected results, such as the component returning an undefined value when implementing `cmp.find("myComponent")`.

lightning:tabset (Beta)

When you load more tabs than can fit the width of the viewport, the tabset provides navigation buttons that scrolls horizontally to display the overflow tabs.

Event Handling in Base Lightning Components

Base components are lightweight and closely resemble HTML markup. They follow standard HTML practices by providing event handlers as attributes, such as `onfocus`, instead of registering and firing Aura component events, like components in the `ui` namespace.

Because of their markup, you might expect to access DOM elements via `event.target` or `event.currentTarget`. However, this type of access breaks encapsulation because it provides access to another component's DOM elements, which are subject to change.

LockerService, which will be enabled for all orgs in Summer '17, enforces encapsulation. Use the methods described here to make your code compliant with LockerService.

To retrieve the component that fired the event, use `event.getSource()`.

```
<aura:component>
    <lightning:button name="myButton" onclick="{!c.doSomething}"/>
</aura:component>
```

```
({
    doSomething: function(cmp, event, helper) {
        var button = event.getSource();

        //The following patterns are not supported
        //when you're trying to access another component's
        //DOM elements.
        var el = event.target;
        var currentEl = event.currentTarget;
    }
})
```

Retrieve a component attribute that's passed to the event by using this syntax.

```
event.getSource().get("v.name")
```

Reusing Event Handlers

`event.getSource()` helps you determine which component fired an event. Let's say you have several buttons that reuse the same `onclick` handler. To retrieve the name of the button that fired the event, use `event.getSource().get("v.name")`.

```
<aura:component>
    <lightning:button label="New Record" name="new" onclick="{!c.handleClick}"/>
    <lightning:button label="Edit" name="edit" onclick="{!c.handleClick}"/>
    <lightning:button label="Delete" name="delete" onclick="{!c.handleClick}"/>
</aura:component>
```

```
({
    handleClick: function(cmp, event, helper) {
        //returns "new", "edit", or "delete"
        var buttonName = event.getSource().get("v.name");
    }
})
```

Retrieving the Active Component Using the `onactive` Handler

Components, such as `lightning:tab` and `lightning:menuItem`, support the `onactive` handler so that you can obtain a reference to the target component when it becomes active. Clicking the component multiple times invokes the handler once only.

For example, you can toggle a check mark on a menu item in a `lightning:buttonMenu` component when it's clicked.

```
<aura:component>
    <lightning:buttonMenu alternativeText="Show menu">
        <lightning:menuItem value="new" onactive="{! c.handleActive }" label="New"
checked="true" />
        <lightning:menuItem value="edit" onactive="{! c.handleActive }" label="Edit"
checked="false" />
        <lightning:menuItem value="delete" onactive="{! c.handleActive }" label="Delete"
checked="false" />
    </lightning:buttonMenu>
</aura:component>
```

```
({
    handleActive: function (cmp, event) {
        var menuItem = event.getSource();
        menuItem.set("v.checked", !menuItem.get("v.checked"));
    }
})
```

Note: If you only need the ID or value of the tab or menu item and you don't need a reference to the target component, use the `onselect` event handler.

Retrieving the ID and Value Using the `onselect` Handler

Some components provide event handlers to pass in events to child components, such as the `onselect` event handler on the following components.

- `lightning:buttonMenu`
- `lightning:tabset`

Although the `event.detail` syntax continues to be supported, we recommend that you update your JavaScript code to use the following patterns for the `onselect` handler as we plan to deprecate `event.detail` in a future release.

- `event.getParam("id")`
- `event.getParam("value")`

For example, you want to retrieve the value of a selected menu item in a `lightning:buttonMenu` component from a client-side controller.

```
//Before
var menuItem = event.detail.menuItem;
var itemValue = menuItem.get("v.value");
//After
var itemValue = event.getParam("value");
```

Similarly, to retrieve the ID of a selected tab in a `lightning:tabset` component:

```
//Before
var tab = event.detail.selectedTab;
var tabId = tab.get("v.id");
//After
var tabId = event.getParam("id");
```

Note: If you need a reference to the target component, use the `onactive` event handler instead.

Working with UI Components

The framework 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. User interface components such as `ui:input` and `ui:output` provide easy handling of common user interface events like keyboard and mouse interactions. Each component can be styled and extended accordingly.

Note: If you are looking for components that apply the Lightning Design System styling, consider using the base lightning components instead.

Complex, Interactive Components

The following components contain one or more sub-components and are interactive.

Type	Key Components	Description
Autocomplete	`ui:autocomplete`	An input field that suggests a list of values as you type
Carousel	`ui:carousel`	A list of pages that can be swiped horizontally
	`ui:carouselPage`	A scrollable page in a `ui:carousel` component
Dialog	`ui:panel`	A modal or non-modal overlay
	`ui:panelManager2`	A component that instantiates and handles panels
Message	`ui:message`	A message notification of varying severity levels

Type	Key Components	Description
Menu	`ui:menu`	A drop-down list with a trigger that controls its visibility. This component extends `ui:popup`.
	`ui:menuList`	A list of menu items
	`ui:actionMenuItem`	A menu item that triggers an action
	`ui:checkboxMenuItem`	A menu item that supports multiple selection and can be used to trigger an action
	`ui:radioMenuItem`	A menu item that supports single selection and can be used to trigger an action
	`ui:menuItemSeparator`	A visual separator for menu items
	`ui:menuItem`	An abstract and extensible component for menu items in a `ui:menuList` component
	`ui:menuTrigger`	A trigger that expands and collapses a menu
	`ui:menuTriggerLink`	A link that triggers a dropdown menu. This component extends `ui:menuTrigger`
Popup	`ui:popup`	A popup with a trigger that controls its visibility. Used by `ui:menu`
	`ui:popupTarget`	A container that's displayed in response to a trigger.
	`ui:popupTrigger`	A trigger that expands and collapses a menu.
Tabset	`ui:tab`	A single tab in a `ui:tabset` component
	`ui:tabBar`	A list wrapper for tabs in a `ui:tabset` component
	`ui:tabItem`	A single tab that's rendered by a `ui:tabBar` component
	`ui:tabset`	A set of tabs that's displayed in an unordered list

Input Control Components

The following components are interactive, for example, like buttons and checkboxes.

Type	Key Components	Description
Button	`ui:button`	An actionable button that can be pressed or clicked
Checkbox	`ui:inputCheckbox`	A selectable option that supports multiple selections
	`ui:outputCheckbox`	Displays a read-only value of the checkbox
Radio button	`ui:inputRadio`	A selectable option that supports only a single selection
Drop-down List	`ui:inputSelect`	A drop-down list with options
	`ui:inputSelectOption`	An option in a `ui:inputSelect` component
	`ui:inputSelectOptionGroup`	

Visual Components

The following components provides informative cues, for example, like error messages and loading spinners.

Type	Key Components	Description
Field-level error	`ui:inputDefaultError`	An error message that is displayed when an error occurs
Input Label	`ui:label`	A text label that binds to an input component
Layout	`ui:block`	A horizontal layout that provides two or three columns
	`ui:vbox`	A vertical layout that provides two or three rows
List	`ui:list`	A collection of items that can be iterated over and displayed
Spinner	`ui:spinner`	A loading spinner

Field Components

The following components enables you to enter or display values.

Type	Key Components	Description
Currency	`ui:inputCurrency`	An input field for entering currency
	`ui:outputCurrency`	Displays currency in a default or specified format
Email	`ui:inputEmail`	An input field for entering an email address
	`ui:outputEmail`	Displays a clickable email address
Date and time	`ui:inputDate`	An input field for entering a date
	`ui:inputDateTime`	An input field for entering a date and time
	`ui:outputDate`	Displays a date in the default or specified format
	`ui:outputDateTime`	Displays a date and time in the default or specified format
Password	`ui:inputSecret`	An input field for entering secret text
Percentage	`ui:inputPercent`	An input field for entering a percentage
	`ui:outputPercent`	Displays a percentage in the default or specified format
Phone Number	`ui:inputPhone`	An input field for entering a telephone number
	`ui:outputPhone`	Displays a phone number
Number	`ui:inputNumber`	An input field for entering a numerical value
	`ui:outputNumber`	Displays a number
Range	`ui:inputRange`	An input field for entering a value within a range
Rich Text	`ui:inputRichText`	An input field for entering rich text

Type	Key Components	Description
	`ui:outputRichText`	Displays rich text
Search	`ui:inputSearch`	An input field for entering a search string
Text	`ui:inputText`	An input field for entering a single line of text
	`ui:outputText`	Displays text
Text Area	`ui:inputTextArea`	An input field for entering multiple lines of text
	`ui:outputTextArea`	Displays a read-only text area
URL	`ui:inputURL`	An input field for entering a URL
	`ui:outputURL`	Displays a clickable URL

SEE ALSO:

Using the UI Components

Creating Components

Component Bundles

Event Handling in UI Components

UI components provide easy handling of user interface events such as keyboard and mouse interactions. By listening to these events, you can also bind values on UI input components using the `updateon` attribute, such that the values update when those events are fired.

Capture a UI event by defining its handler on the component. For example, you want to listen to the HTML DOM event, `onblur`, on a `ui:inputTextArea` component.

```
<ui:inputTextArea aura:id="textarea" value="My text area" label="Type something"
      blur="{!c.handleBlur}" />
```

The `blur="{!c.handleBlur}"` listens to the `onblur` event and wires it to your client-side controller. When you trigger the event, the following client-side controller handles the event.

```
handleBlur : function(cmp, event, helper){
    var elem = cmp.find("textarea").getElement();
    //do something else
}
```

These events are available to any components that implement the `ui:visible` and `ui:uiEvents` interfaces. The `ui:visible` interface provides event registration for mouse events and attributes that defines a component's class and label. The `ui:uiEvents` interface provides event registration for form events, such as `blur` and `focus`.

For all available events on all components, refer to the Reference Doc App on page 259.

Value Binding for Browser Events

Any changes to the UI are reflected in the component attribute, and any change in that attribute is propagated to the UI. When you load the component, the value of the input elements are initialized to those of the component attributes. Any changes to the user input causes the value of the component variable to be updated. For example, a `ui:inputText` component can contain a value that's bound to a component attribute, and the `ui:outputText` component is bound to the same component attribute. The `ui:inputText` component listens to the `onkeyup` browser event and updates the corresponding component attribute values.

```
<aura:attribute name="first" type="String" default="John"/>
<aura:attribute name="last" type="String" default="Doe"/>

<ui:inputText label="First Name" value="{!v.first}" updateOn="keyup"/>
<ui:inputText label="Last Name" value="{!v.last}" updateOn="keyup"/>

<!-- Returns "John Doe" -->
<ui:outputText value="{!v.first +' '+ v.last}"/>
```

The next example takes in numerical inputs and returns the sum of those numbers. The `ui:inputNumber` component listens to the `onkeyup` browser event. When the value in this component changes on the keyup event, the value in the `ui:outputNumber` component is updated as well, and returns the sum of the two values.

```
<aura:attribute name="number1" type="integer" default="1"/>
<aura:attribute name="number2" type="integer" default="2"/>

<ui:inputNumber label="Number 1" value="{!v.number1}" updateOn="keyup" />
<ui:inputNumber label="Number 2" value="{!v.number2}"  updateOn="keyup" />

<!-- Adds the numbers and returns the sum -->
<ui:outputNumber  value="{!(v.number1 * 1) + (v.number2 * 1)}"/>
```

Note: The input fields return a string value and must be properly handled to accommodate numerical values. In this example, both values are multiplied by 1 to obtain their numerical equivalents.

Using the UI Components

Users interact with your app through input elements to select or enter values. Components such as `ui:inputText` and `ui:inputCheckbox` correspond to common input elements. These components simplify event handling for user interface events.

Note: For all available component attributes and events, see the component reference at `http://<myServer>/auradocs/reference.app`.

To use input components in your own custom component, add them to your `.cmp` or `.app` file. This example is a basic set up of a text field and button. The `aura:id` attribute defines a unique ID that enables you to reference the component from your JavaScript code using `cmp.find("myID");`.

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

Note: All text fields must specify the `label` attribute to provide a textual label of the field. If you must hide the label from view, set `labelClass="assistiveText"` to make the label available to assistive technologies.

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);
    }
```

The following example is similar to the previous, but uses value binding without a client-side controller. The `ui:outputText` component reflects the latest value on the `ui:inputText` component when the `onkeyup` browser event is fired.

```
<aura:attribute name="first" type="String" default="John"/>
<aura:attribute name="last" type="String" default="Doe"/>

<ui:inputText label="First Name" value="{!v.first}" updateOn="keyup"/>
<ui:inputText label="Last Name" value="{!v.last}" updateOn="keyup"/>

<!-- Returns "John Doe" -->
<ui:outputText value="{!v.first +' '+ v.last}"/>
```

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`. Provide a date picker by setting `displayDatePicker="true"`. Web apps running on mobiles and tablets use an input field of type `date`.	`ui:inputDate` `ui:outputDate`
Date and Time	An input field for entering a date and time of type `text`. Provide a date picker and time picker by setting `displayDatePicker="true"`. On desktop, the date and time fields display as two separate fields. The time picker displays a list of time in 30-minute increments. Web apps running on mobiles and tablets use an input field of type `datetime-local`.	`ui:inputDateTime` `ui:outputDateTime`

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 uiInput--default uiInput--input uiInput--datetime">
  <label class="uiLabel-left form-element__label uiLabel">
    <span>Birthday</span>
  </label>
  <form class="form--stacked form-element">
      <input placeholder="MMM d, yyyy" type="text">
      <a class="datePicker-openIcon display" aria-haspopup="true">
          <span class="assistiveText">Date Picker</span>
      </a>
  <a class="clearIcon hide">
    <span class="assistiveText">Clear Button</span>
  </a>
  </form>
</div>
<div class="DESKTOP uiDatePicker--default uiDatePicker">
    <!--Date picker set to visible when icon is clicked-->
</div>
```

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.

Type	Related Components	Description
Number	`ui:inputNumber`	An input field for entering a numerical value
	`ui:outputNumber`	Displays a number
Currency	`ui:inputCurrency`	An input field for entering currency
	`ui:outputCurrency`	Displays currency

Type	Related Components	Description
Percentage	`ui:inputPercent` `ui:outputPercent`	An input field for entering a numerical percentage value.
Range	`ui:inputRange`	A slider for numerical input.

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>
<iput aria-describedby max="99999999999999" step="1" placeholder type="text"
min="-99999999999999" 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}"/>
```

Note: `{#v.number}` is an unbound expression. This means that any change to the `value` attribute in `ui:inputNumber` doesn't propagate back to affect the value of the `number` attribute in the parent component. For more information, see Data Binding Between Components on page 27.

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:inputNumber label="Cost" 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:outputPercent label="Growth" 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" />
```

Number and Currency Shortcuts

Users can enter the shortcuts **k**, **m**, **b**, **t** to indicate thousands, millions, billions, or trillions in `ui:inputNumber` and `ui:inputCurrency` components. This feature is available in Lightning Experience, and all versions of the Salesforce1 mobile app.

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`, or `.THIS.uiInputCurrency` if it's a top-level element.

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 provide 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.

Type	Related Components	Description
Email	`ui:inputEmail`	An input field for entering an email address
	`ui:outputEmail`	Displays a clickable email address
Password	`ui:inputSecret`	An input field for entering secret text
Phone Number	`ui:inputPhone`	An input field for entering a telephone number
	`ui:outputPhone`	Displays a clickable phone number
Rich Text	`ui:inputRichText`	An input field for entering rich text
	`ui:outputRichText`	Displays rich text
Search	`ui:inputSearch`	An input field for entering a search term.
Text	`ui:inputText`	An input field for entering single line of text
	`ui:outputText`	Displays text
Text Area	`ui:inputTextArea`	An input field for entering multiple lines of text
	`ui:outputTextArea`	Displays a read-only text area
URL	`ui:inputURL`	An input field for entering a URL
	`ui:outputURL`	Displays a clickable URL

Using the Text Fields

Text fields are typically used in a form. For example, 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 uiInputEmail uiInput--default uiInput--input">
  <label class="uiLabel-left form-element__label uiLabel">
    <span>Email</span>
  </label>
  <input placeholder="abc@email.com" type="email" class="input">
</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.

For example, to style the `ui:inputPhone` component, use `.THIS .uiInputPhone`, or `.THIS.uiInputPhone` if it's a top-level element.

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

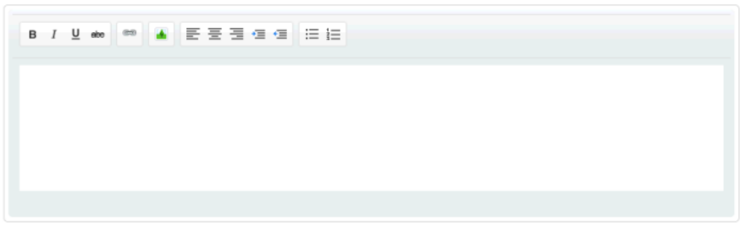
Rich Text Fields

`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"
   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"
    buttonTitle="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" />
```

The previous example results in the following HTML.

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

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" />

    /*The controller action*/
    crossout : function(cmp, event){
        var cmpSource = event.getSource();
        $A.util.toggleClass(cmpSource, "done");
     }
```

SEE ALSO:

Java Models

Handling Events with Client-Side Controllers

CSS in Components

Radio Buttons

Radio buttons are clickable and actionable, and they can only be individually selected when presented in a group. You can create a radio button with `ui:inputRadio`, which inherits the behavior and events from `ui:input`. The `value` and `disabled` attributes control the state of a radio button, and events such as `click` and `change` determine its behavior. Events must be used separately on each radio button.

If you want to use radio buttons in a menu, use `ui:radioMenuItem` instead.

Here are several basic ways to set up a radio button.

Selected

To select the radio button, set `value="true"`.

```
<ui:inputRadio value="true" label="Select?"/>
```

Disabled State

```
<ui:inputRadio label="Select" disabled="true"/>
```

The previous example results in the following HTML.

```
<div class="uiInput uiInputRadio uiInput--default uiInput--radio">
    <label class="uiLabel-left form-element__label uiLabel"
for="globalId"><span>Select</span></label>
<input type="radio" id="globalId">
```

Providing Labels using An Attribute

You can also initialize the label values using an attribute. This example uses an attribute to populate the radio button labels and wire them up to a client-side controller action when the radio button is selected or deselected.

```
<!--c:labelsAttribute-->
<aura:component>
    <aura:attribute name="stages" type="String[]" default="Any,Open,Closed,Closed,Closed
Won"/>
    <aura:iteration items="{#v.stages}" var="stage">
        <ui:inputRadio label="{#stage}" change="{!c.doSomething}"/>
    </aura:iteration>
</aura:component>
```

Note: `{#v.stages}` and `{#stage}` are unbound expressions. This means that any change to the value of the `items` attribute in `aura:iteration` or the `label` attribute in `ui:inputRadio` don't propagate back to affect the value of the `stages` attribute in `c:labelsAttribute`. For more information, see Data Binding Between Components on page 27.

Working with Events

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

The following code updates the CSS class of a component when the radio button is clicked.

```
<!--The radio button-->
    <ui:inputRadio click="{!c.showItem}" label="Show Item"/>
```

```
/* The controller action */
showItem : function(cmp, event) {
    var myCmp = cmp.find('myCmp');
    $A.util.toggleClass(myCmp, "cssClass");
}
```

SEE ALSO:

Handling Events with Client-Side Controllers

CSS in Components

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="button uiButton--default uiButton" accesskey type="button">
  <img class="icon bLeft" alt="Find" src="path/to/img">
  <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}"/>
  <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

Which Button Was Pressed?

Drop-down Lists

Drop-down lists display a dropdown menu with options you can select.

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, set the `multiple` attribute to `true`.

Single Selection

```
<ui:inputSelect>
        <ui:inputSelectOption text="Red"/>
        <ui:inputSelectOption text="Green" value="true"/>
        <ui:inputSelectOption text="Blue"/>
</ui:inputSelect>
```

Multiple Selection

```
<ui:inputSelect multiple="true">
    <ui:inputSelectOption text="All Primary" label="All Contacts"/>
    <ui:inputSelectOption text="All Primary" label="All Primary"/>
    <ui:inputSelectOption text="All Secondary" label="All Secondary"/>
</ui:inputSelect>
```

Each option is represented by `ui:inputSelectOption`. The default selected value is specified by `value="true"` on the option.

Note: `v.value` represents the option's HTML `selected` attribute, and `v.text` represents the option's HTML `value` attribute.

Generating Options with `aura:iteration`

You can use `aura:iteration` to iterate over a list of items to generate options. This example iterates over a list of items and conditionally renders the options.

```
<aura:attribute name="contacts" type="String[]" default="All Contacts,Others"/>
<ui:inputSelect>
    <aura:iteration items="{!v.contacts}" var="contact">
        <aura:if isTrue="{!contact == 'All Contacts'}">
            <ui:inputSelectOption text="{!contact}" label="{!contact}"/>
            <aura:set attribute="else">
                <ui:inputSelectOption text="All Primary" label="All Primary"/>
                <ui:inputSelectOption text="All Secondary" label="All Secondary"/>
            </aura:set>
        </aura:if>
    </aura:iteration>
</ui:inputSelect>
```

Generating Options Dynamically

Generate the options dynamically on component initialization.

```
<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 `v.options` on the `ui:inputSelect` component by creating the `opts` object with several parameters. `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"},
            { class: "optionClass", label: "Option2", value: "opt2" },
            { class: "optionClass", label: "Option3", value: "opt3" }

        ];
        cmp.find("InputSelectDynamic").set("v.options", opts);
    }
})
```

Note: `class` is a reserved word that might not work with older versions of Internet Explorer. We recommend using `"class"` with double quotes.

The list options support these parameters.

Parameter	Type	Description
class	String	The CSS class for the option.
disabled	Boolean	Indicates whether the option is disabled.

Parameter	Type	Description
label	String	The label of the option to display on the user interface.
selected	Boolean	Indicates whether the option is selected.
value	String	Required. The value of the option.

Using Options On Multiple Lists

If you're reusing the same set of options on multiple drop-down lists, use different attributes for each set of options. Otherwise, selecting a different option in one list also updates other list options bound to the same attribute.

```
<aura:attribute name="options1" type="String" />
<aura:attribute name="options2" type="String" />
<ui:inputSelect aura:id="Select1" label="Select1" options="{!v.options1}" />
<ui:inputSelect aura:id="Select2" label="Select2" options="{!v.options2}" />
```

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;
    height: 100px;
}
```

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

SEE ALSO:

Handling Events with Client-Side Controllers

CSS in Components

Field-level Errors

Field-level errors are displayed when an input validation error occurs on the field. Input components in the `lightning` namespace display a default error message when an error condition is met. For input components in the `ui` namespace, the framework 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 by toggling visibility of the message when an error condition is met. See Dynamically Showing or Hiding Markup for more information.

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 the error message.

SEE ALSO:

Handling Events with Client-Side Controllers

CSS in Components

Menus

A menu is a drop-down list with a trigger that controls its visibility. You must provide the trigger and list of menu items. The dropdown menu and its menu items are hidden by default. You can change this by setting the `visible` attribute on the `ui:menuList` component to `true`. The menu items are shown only when you click the `ui:menuTriggerLink` component.

This example creates a menu with several items.

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Opportunity Status"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu">
              <ui:actionMenuItem aura:id="item2" label="Open"
click="{!c.updateTriggerLabel}"/>
              <ui:actionMenuItem aura:id="item3" label="Closed"
click="{!c.updateTriggerLabel}"/>
              <ui:actionMenuItem aura:id="item4" label="Closed Won"
click="{!c.updateTriggerLabel}"/>
        </ui:menuList>
</ui:menu>
```

The following components are nested in `ui:menu`.

Component	Description
`ui:menu`	A drop-down list with a trigger that controls its visibility
`ui:menuList`	A list of menu items
`ui:actionMenuItem`	A menu item that triggers an action
`ui:checkboxMenuItem`	A menu item that supports multiple selection and can be used to trigger an action
`ui:radioMenuItem`	A menu item that supports single selection and can be used to trigger an action
`ui:menuItemSeparator`	A visual separator for menu items
`ui:menuItem`	An abstract and extensible component for menu items in a `ui:menuList` component
`ui:menuTrigger`	A trigger that expands and collapses a menu
`ui:menuTriggerLink`	A link that triggers a dropdown menu. This component extends `ui:menuTrigger`

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.

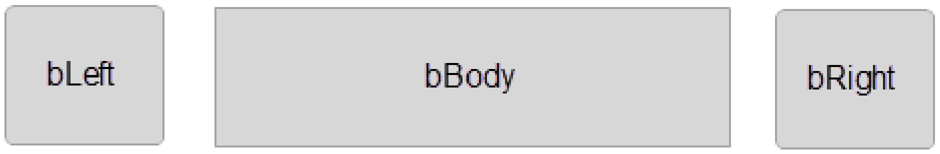
```
<ui:block>
    <aura:set attribute="left">
        <ui:image src="/auraFW/resources/aura/images/search.png" alt="bLeft" />
    </aura:set>
    <aura:set attribute="right">
        <ui:button label="Submit"/>
    </aura:set>
    <ui:inputText label="Text" labelPosition="hidden" />
</ui:block>
```

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.

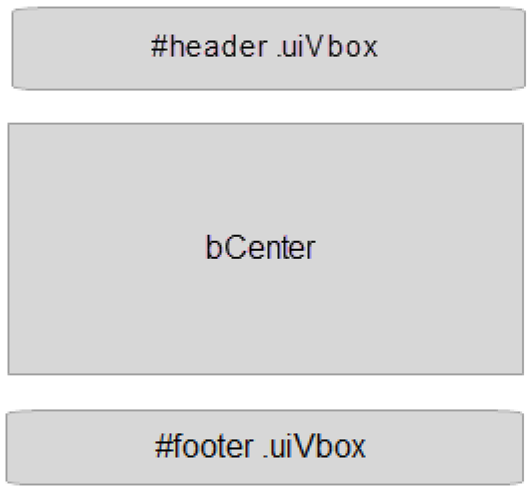
```
<ui:vbox>
    <aura:set attribute="north">
    <div id="header">Header</div>
    </aura:set>
    <aura:set attribute="south">
    <div id="footer">Footer</div>
    </aura:set>
    body
</ui:vbox>
```

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 fuction 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(response) {
    var state = response.getState();
    if (state === "SUCCESS") {
        var result = response.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

Supporting Accessibility

Components are created with accessibility in mind. This is also true for components that extend these components.

When customizing components, be careful in preserving code that ensures accessibility, such as the `aria` attributes.

Accessible software and assistive technology enable users with disabilities to use and interact with the products you build. Aura components are created according to W3C specifications so that they work with common assistive technologies. While we always recommend that you follow the WCAG Guidelines for accessibility when developing with Aura, this guide explains the accessibility features that you can leverage when using components in the `ui` namespace.

In general, you can think of the components as either basic or complex, interactive components.

Basic Components

- `ui:image` — for images and icons
- `ui:input` — for input elements such as text fields and date fields
- `ui:button` — for input elements such as push buttons, radio buttons, and checkboxes

Complex, Interactive Components

- `ui:autocomplete` — for autocompleting dropdowns
- `ui:carousel` — for carousel interactions
- `ui:tabset` — for tab and tab panel interactions
- `ui:datePicker` — for calendar pickers
- `ui:panel` — for modal and non-modal overlays
- `ui:menu` — for menus, dropdowns, and muttons
- `ui:message` — for displaying page updates to users and updating screen readers

Accessibility Testing

To check that a component's HTML output is compliant with our accessibility validation, run `$A.test.assertAccessible()`. You can also run `$A.devToolService.checkAccessibility()` on a browser console. This tool checks the rendered DOM elements to make sure that they pass Salesforce's accessibility validation. Examples of this include image tags requiring an `alt` attribute, active panels correctly setting the `aria-hidden` attribute, and `input`, `select`, and `textarea` tags having associated labels.

When using the tool, there are two outcomes: pass or fail. If the tool does not find any accessibility exceptions, it returns an empty string. When the tool does find accessibility exceptions, it will include the accessibility rule that failed, the erroneous tag, and a stacktrace of where it was found in the code.

To use these tests, you must have the Aura Framework loaded. The tests can be used in the console (`$A.devToolService.checkAccessibility()`), JSTEST (`$A.test.assertAccessible()`), or in a WebDriver test (`auraTestingUtil.assertAccessible()`).

The tests look for these issues:

- Images without the `alt` attribute
- Anchor element without textual content
- `input` elements without an associated label
- Radio button groups 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

Since Aura is a single page javascript application, the person writing the test will have to make sure to re-test when the DOM changes. The person using the tool should place a check after the DOM has changed to ensure greater accessibility validation coverage.

The sections below include more information specific to different types of components.

IN THIS SECTION:

Button Labels

Buttons may be designed to appear with just text, an image and text, or an image without 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 visible on screen.

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

When using `ui:button`, assign a non-empty string to label attribute. If it's an icon only button, use `labelDisplay` in `ui:button` to hide the label text. These examples show how a `ui:button` should render:

```
<!-- Good: using alt attribute to provide a invisible label -->
<button>
    <img src="search.png" alt="Search"/>
</button>
```

```
<!-- Good: using span/assistiveText to hide the label visually, but show it to screen
readers -->
<button>
 ::before
    <span class="assistiveText">Search</span>
</button>
```

SEE ALSO:

Buttons

Carousels

The `ui:carousel` component displays a list of items horizontally where users can swipe through the list or click through the page indicators.

If your code failed, check to make sure the page indicators are visible. If `visible="false"` is set on the `ui:carouselPageIndicatorItem`, the page indicators will be hidden 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. Let's say you want to create help text for `ui:inputText`.

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

Using the input component to create and handle the `ui:inputDefaultError` component automatically applies the `ariaDescribedby` attribute on the error messages. If you want to manually manage the action, you will need to make the connection between the `ui:inputDefaultError` component and the associated output.

Your component should render like this example:

```
<!-- Good: aria-describedby is used to associate error message -->
<label for="fname">Contact name</label>
<input name="" type="text" id="fname" aria-describedby="msgid">
<ul class="uiInputDefaultError" id="msgid">
    <li>Please enter the contact name</li>
</ul>
```

SEE ALSO:

Validating Fields

Audio Messages

To convey audio notifications, use the `ui:message` component, which has `role="alert"` set on the component by default. The `"alert"` `aria` role will take any text inside the div and read it out loud to screen readers without any additional action by the user.

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

Forms, Fields, and Labels

Input components are designed to make it easy to assign labels to form fields. Labels build a programmatic relationship between a form field and its textual label. 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." />
```

If your code failed, check the label element during component rendering. A label element should have the `for` attribute and match the value of input control id attribute, OR the label should be wrapped around an input. Input controls include `<input>`, `<textarea>`, and `<select>`.

```
<!-- Good: using label/for= -->
<label for="fullname">Enter your full name:</label>
<input type="text" id="fullname" />

<!-- Good: --using implicit label>
<label>Enter your full name:
    <input type="text" id="fullname"/>
</label>
```

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>
```

IN THIS SECTION:

Using Images

Using Images

To display images, use the `ui:image` component. The `ui:image` component automates common usages of the HTML `<img>` tag, such as `href` linking and other attributes. Additionally, include the `imageType` attribute to show 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 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>
```

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" />
```

Code Samples

If your code failed, check to make sure you used the `alt` tag and the `assistiveText` class correctly.

Informational image code example:

```
alt tag:
<ui:image src="follow.png" imageType="informational" alt="follow" />
assistiveText class:
<div class="Following">
    <span class="assistiveText">Following</span>
</div>
```

Decorative image code example:

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

Events

Although you can attach an `onclick` event to any type of element, for accessibility, consider only applying this event to elements that are actionable in HTML by default, such as `<a>`, `<button>`, or `<input>` tags in component markup. You can use an `onclick` event on a `<div>` tag to prevent event bubbling of a click.

Dialog Overlays

The `ui:panel` component creates an overlay that lets users access additional information without leaving the current page. Modal overlay requires the user to take an action or cancel the overlay to go back to the original page. Non-modal overlays offer useful information but can be ignored by users.

To create a modal overlay, use `panelType: 'modal'`, which locks keyboard focus inside the modal. `autoFocus` must be also true for the component to be accessible. `autoFocus` is true by default. To create a non-modal overlay, set `panelType: 'panel'` and users can just tab through. Fire the `ui:createPanel` event to create a modal or non-modal overlay.

```
$A.get('e.ui:createPanel').setParams({
    panelType: 'modal',
    visible: true,
    panelConfig: {
        title: 'Modal Header',
        autoFocus: true,
        body: body,
        footer: footer
        },
        onCreate: function(panel){
            //do something
        }
    }).fire();
```

`ui:panel` needs to have a title to meet accessibility standards, but it doesn't have to be visible. Use `titleDisplay: false` to hide the title, if desired.

Menus

A menu is a drop-down list with a trigger that controls its visibility. You must provide the trigger and list of menu items. The drop-down menu and its menu items are hidden by default. You can change this by setting the `visible` attribute on the `ui:menuList` component to `true`. The menu items are shown only when you click the `ui:menuTriggerLink` component.

This example code creates a menu with several items:

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Opportunity Status"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu">
              <ui:actionMenuItem aura:id="item2" label="Open"
click="{!c.updateTriggerLabel}"/>
              <ui:actionMenuItem aura:id="item3" label="Closed"
click="{!c.updateTriggerLabel}"/>
              <ui:actionMenuItem aura:id="item4" label="Closed Won"
click="{!c.updateTriggerLabel}"/>
        </ui:menuList>
</ui:menu>
```

Different menus achieve different goals. Make sure you use the right menu for the desired behavior. The three types of menus are:

Actions
Use the `ui:actionMenuItem` for items that create an action, like print, new, or save.

Radio button
If you want users to pick only one from a list several items, use `ui:radioMenuItem`.

Checkbox style

If users can pick multiple items from a list of several items, use `ui:checkboxMenuItem`. Checkboxes can also be used to turn one item on or off.

Resolving Accessibility Errors

Accessibility tests validate generated HTML markup and may return an error code followed by a message to help you resolve those errors.

The following errors flag accessibility issues in your components. Resolve these errors to ensure that your components are accessible.

[A11Y_DOM_01] All image tags require the presence of the alt attribute

Informational images must have a description set on its `alt` attribute. If the image is decorative, set `alt=""`. For more information, see Images on page 86.

```
<!-- Informational image -->
<img src="admin.png" alt="admin image">
```

[A11Y_DOM_02] Labels are required for all input controls

A label element should have a `for` attribute and match the value of the `id` attribute on the input control, or the label should be wrapped around the input. Input controls include `<input>`, `<textarea>` and `<select>`. For more information, see Forms, Fields, and Labels on page 85.

```
<!-- Method 1: Use label/for -->
<label for="fullname">Enter your full name:</label>
<input type="text" id="fullname"/>

 <!-- Method 2: Use an implicit label-->
<label>Enter your full name:
    <input type="text" id="fullname"/>
</label>
```

[A11Y_DOM_03] Buttons must have non-empty text labels

When using `ui:button`, assign a non-empty string to the `label` attribute. For an icon-only button, use `labelDisplay` in `ui:button` to hide the label text. For more information, see Button Labels on page 84.

```
<!-- Method 1: Use the alt attribute to provide a hidden label -->
<button>
    <img src="enter_site.gif" alt="enter site"/>
</button>

<!-- Method 2: Use a span tag with assistiveText class to hide the label visually -->
<button>
    <span class="assistiveText">Enter site</span>
</button>
```

[A11Y_DOM_04] Links must have non-empty text content

For a graphical link, use a `ui:image` instead. To include hidden link text, use a `span` tag with `assistiveText` class. For buttons, use the `ui:button` component.

```
<!-- Method 1: Use an img tag with the alt attribute to provide link text  -->
<a href="routes.html">
    <img src="routes.png" alt="current routes" />
</a>
```

```
<!-- Method 2: Use a span tag with assistiveText class to provide link text -->
<a href="javascript:void(0);">
    <span class="assistiveText">Toggle Notifications</span>
    <div class="notificationCounter"></div>
</a>
```

[A11Y_DOM_05] Text color contrast ratio must meet the minimum requirement

Small text must have a contrast ratio of not less than 4.5:1. Small text includes those whose font size are:

- Smaller than 19px bold or semibold
- Smaller than 24px normal

Large text must have a contrast ratio of not less than 3.0:1. Large text includes those whose font size are:

- At least 19px bold or semibold
- At least 24px normal

A good color contrast ratio means that the foreground and background color provides enough contrast when viewed by a user who might have impaired vision or when viewed on a black and white screen. You can install Accessibility Developer Tools on your Google Chrome browser or use the WebAim Color Contrast Checker tool.

[A11Y_DOM_06] Each frame and iframe element must have a non-empty title attribute

If using an `iframe` element, include a descriptive `title` attribute.

```
<iframe src="banner-ad.html" id="testiframe" name="testiframe" title="Advertisement">
    <a href="banner-ad.html">Advertisement</a>
</iframe>
```

[A11Y_DOM_07] The head section must have a non-empty title element

In the `head` element, include a descriptive `title` tag.

```
<head>
    <title>Welcome</title>
</head>
```

[A11Y_DOM_08] Data table cells must be associated with data table headers

Use the `scope` attribute or use both the `id` and `header` attributes.

```
<!-- Method 1: Use the scope attribute -->
<table border="1"><caption>Contact Information</caption>
    <tr>
        <th scope="col">Name</th>
        <th scope="col">Department</th>
    </tr>
    <tr>
        <td>admin</td>
        <td>R&D</td>
    </tr>
</table>

<!-- Method 2: Use the id and headers attributes -->
<table border="1">
    <tr>
        <th id="e1">First Name</th>
        <th id="e2">Last Name</th>
        <th id="e3">Department</th>
    </tr>
    <tr>
        <td headers="e1">John</td>
        <td headers="e2">Smith</td>
        <td headers="e3">R&D</td>
    </tr>
</table>
```

[A11Y_DOM_09] Fieldset must have a legend element

Include a descriptive legend in your `fieldset` element.

```
<fieldset>
    <legend>Choose yes or no</legend>
</fieldset>
```

[A11Y_DOM_10] Related radio buttons or checkboxes must be grouped with a fieldset

Nest your radio buttons and checkboxes in a `fieldset` tag.

```
<fieldset>
    <legend>Choose yes or no</legend>
    <input type="radio" name="yes" id="yesid" value="yes"/>
    <label for="yesid">yes</label>
    <input type="radio" name="no" id="noid" value="no"/>
    <label for="noid">no</label>
</fieldset>
```

[A11Y_DOM_11] Headings should be properly nested

Headings should increase no more than one level each time, and can start at any level.

```
<h2>Profile</h2>
<h3>Profile Details</h3>
<h2>Interests</h2>
```

[A11Y_DOM_12] Base and top panels should have proper aria-hidden properties

The `aria-hidden` attribute indicates whether an element is hidden or not, and can be set to `true` or `false` respectively.

```
<!-- aria-hidden of base panel is false if top panel is not active -->
<section class="stage panelSlide forceAccess" aria-hidden="false"></div>
<div class="panel panelOverlay" aria-hidden="true"></div>

<!-- aria-hidden of base panel is true if there is active top panel -->
<section class="stage panelSlide forceAccess" aria-hidden="true"></div>
<div class="panel panelOverlay active" aria-hidden="false"></div>
```

[A11Y_DOM_13] Aria-describedby must be used to associate error message with input control

The `aria-describedby` attribute indicates the IDs of the elements that describe the object, and can be used to associate static text with groups of elements. For more information, see Help and Error Messages on page 85.

```
<label for="fname">First name</label>
<input name="firstname" type="text" id="fname" aria-describedby="msgid">
<ul class="uiInputDefaultError" id="msgid">
```

Add Components to Apps

When you're ready to add components to your app, you should first look at the out-of-the-box components that come with the framework. 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 out-of-the-box components. 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
- Communicating with Events

CHAPTER 4 Communicating with Events

In this chapter ...

The framework uses event-driven programming. You write handlers that respond to interface events as they occur. The events may or may not have been triggered by user interaction.

In 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.

Events are declared by the `aura:event` tag in a `.evt` file, and they can have one of two types: component or application.

Component Events
: A component event is fired from an instance of a component. A component event can be handled by the component that fired the event or by a component in the containment hierarchy that receives the event.

Application Events
: Application events follow a traditional publish-subscribe model. An application event is fired from an instance of a component. All components that provide a handler for the event are notified.

Note: Always try to use a component event instead of an application event, if possible. Component events can only be handled by components above them in the containment hierarchy so their usage is more localized to the components that need to know about them. Application events are best used for something that should be handled at the application level, such as navigating to a specific record. Application events allow communication between components that are in separate parts of the application and have no direct containment relationship.

Actions and Events

The framework uses events to communicate data between components. Events are usually triggered by a user action.

Actions

User interaction with an element on a component or app. User actions trigger events, but events aren't always explicitly triggered by user actions. This type of action is *not* the same as a client-side JavaScript controller, which is sometimes known as a *controller action*. The following button is wired up to a browser `onclick` event in response to a button click.

```
<ui:button label = "Click Me" press = "{!c.handleClick}" />
```

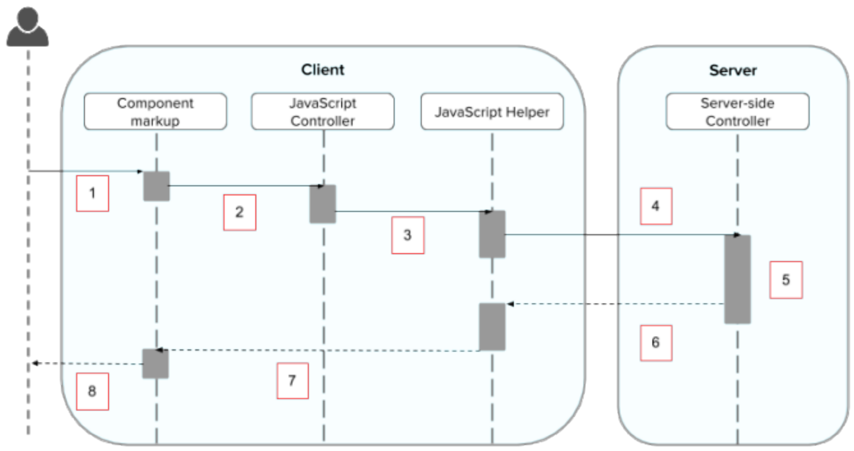
Clicking the button invokes the `handeClick` method in the component's client-side controller.

Events

A notification by the browser regarding an action. Browser events are handled by client-side JavaScript controllers, as shown in the previous example. A browser event is not the same as a framework *component event* or *application event*, which you can create and fire in a JavaScript controller to communicate data between components. For example, you can wire up the click event of a checkbox to a client-side controller, which fires a component event to communicate relevant data to a parent component.

Another type of event, known as a *system event*, is fired automatically by the framework during its lifecycle, such as during component initialization, change of an attribute value, and rendering. Components can handle a system event by registering the event in the component markup.

The following diagram describes what happens when a user clicks a button that requires the component to retrieve data from the server.



1. User clicks a button or interacts with a component, triggering a browser event. For example, you want to save data from the server when the button is clicked.
2. The button click invokes a client-side JavaScript controller, which provides some custom logic before invoking a helper function.
3. The JavaScript controller invokes a helper function. A helper function improves code reuse but it's optional for this example.
4. The helper function calls a server-side controller method and queues the action.
5. The server-side method is invoked and data is returned.
6. A JavaScript callback function is invoked when the server-side method completes.
7. The JavaScript callback function evaluates logic and updates the component's UI.

8. User sees the updated component.

SEE ALSO:

Handling Events with Client-Side Controllers
Detecting Data Changes with Change Handlers
Calling a Server-Side Action
Events Fired During the Rendering Lifecycle

Handling Events with Client-Side Controllers

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

Client-side controllers are surrounded by brackets and curly braces to denote a JSON object containing a map of name-value pairs.

```
({
    myAction : function(cmp, event, helper) {
        // add code for the action
    }
})
```

Each action function takes in three parameters:

1. `cmp`—The component to which the controller belongs.
2. `event`—The event that the action is handling.
3. `helper`—The component's helper, which is optional. A helper contains functions that can be reused by any JavaScript code in the component bundle.

Creating a Client-Side Controller

A client-side controller is part of the component bundle. It is auto-wired via the naming convention, *`componentName`*`Controller.js`.

To 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 `c.sampleComponent` in `c/sampleComponent/sampleComponentController.js`.

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

Calling Client-Side Controller Actions

The following example component creates two buttons to contrast an HTML button with a `<ui:button>`, which is a standard Aura component. Clicking on these buttons updates the `text` component attribute with the specified values. `target.get("v.label")` refers to the `label` attribute value on the button.

Component source

```
<aura:component>
    <aura:attribute name="text" type="String" default="Just a string. Waiting for change."/>

     <input type="button" value="Flawed HTML Button"
         onclick="alert('this will not work')"/>
     <br/>
     <ui:button label="Framework Button" press="{!c.handleClick}"/>
     <br/>
     {!v.text}
</aura:component>
```

If you know some JavaScript, you might be tempted to write something like the first "Flawed" button because you know that HTML tags are first-class citizens in the framework. However, the "Flawed" button won't work because arbitrary JavaScript, such as the `alert()` call, in the component is ignored.

The framework has its own event system. DOM events are mapped to Aura events, since HTML tags are mapped to Aura components.

Any browser DOM element event starting with `on`, such as `onclick` or `onkeypress`, can be wired to a controller action. You can only wire browser events to controller actions.

The "Framework" button wires the `press` attribute in the `<ui:button>` component to the `handleClick` action in the controller.

Client-side controller source

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

        var target = event.getSource();
        cmp.set("v.text", target.get("v.label"));
    }
})
```

The `handleClick` action uses `event.getSource()` to get the source component that fired this component event. In this case, the source component is the `<ui:button>` in the markup.

The code then sets the value of the `text` component attribute to the value of the button's `label` attribute. The `text` component attribute is defined in the `<aura:attribute>` tag in the markup.

Handling Framework Events

Handle framework events using actions in client-side component controllers. Framework events for common mouse and keyboard interactions are available with out-of-the-box components. 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`.

Tip: Use unique names for client-side and server-side actions in a component. A JavaScript function (client-side action) with the same name as a server-side action (Java method) can lead to hard-to-debug issues.

Accessing Component Attributes

In the `handleClick` function, notice that the first argument to every action is the component to which the controller belongs. One of the most common things you'll want to do with this component is look at and change its attribute values.

`cmp.get("v.attributeName")` returns the value of the ***attributeName*** attribute.

`cmp.set("v.attributeName", "attribute value")` sets the value of the ***attributeName*** attribute.

Invoking Another Action in the Controller

To call an action method from another method, put the common code in a helper function and invoke it using `helper.someFunction(cmp)`.

SEE ALSO:

Sharing JavaScript Code in a Component Bundle

Event Handling Lifecycle

Creating Server-Side Logic with Controllers

Component Events

A component event is fired from an instance of a component. A component event can be handled by the component that fired the event or by a component in the containment hierarchy that receives the event.

Note: To communicate from a parent component to a child component that it contains, use `<aura:method>` to call a method in the child component's client-side controller from the parent component. Using `<aura:method>` is easier than getting an instance of the child component in the parent component, and then firing and handling a component event.

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

IN THIS SECTION:

Component Event Propagation

The framework supports *capture* and *bubble* phases for the propagation of component events. These phases are similar to DOM handling patterns and provide an opportunity for interested components to interact with an event and potentially control the behavior for subsequent handlers.

Create Custom Component Events

Create a custom component event using the `<aura:event>` tag in a `.evt` resource. Events can contain attributes that can be set before the event is fired and read when the event is handled.

Fire Component Events

Fire a component event to communicate data to another component. A component event can be handled by the component that fired the event or by a component in the containment hierarchy that receives the event.

Handling Component Events

A component event can be handled by the component that fired the event or by a component in the containment hierarchy that receives the event.

SEE ALSO:

aura:method

Application Events

Handling Events with Client-Side Controllers

Advanced Events Example

What is Inherited?

Component Event Propagation

The framework supports *capture* and *bubble* phases for the propagation of component events. These phases are similar to DOM handling patterns and provide an opportunity for interested components to interact with an event and potentially control the behavior for subsequent handlers.

The component that fires an event is known as the source component. The framework allows you to handle the event in different phases. These phases give you flexibility for how to best process the event for your application.

The phases are:

Capture

The event is captured and trickles down from the application root to the source component. The event can be handled by a component in the containment hierarchy that receives the captured event.

Event handlers are invoked in order from the application root down to the source component that fired the event.

Any registered handler in this phase can stop the event from propagating, at which point no more handlers are called in this phase or the bubble phase.

Bubble

The component that fired the event can handle it. The event then bubbles up from the source component to the application root. The event can be handled by a component in the containment hierarchy that receives the bubbled event.

Event handlers are invoked in order from the source component that fired the event up to the application root.

Any registered handler in this phase can stop the event from propagating, at which point no more handlers are called in this phase.

Here's the sequence of component event propagation.

1. **Event fired**—A component event is fired.
2. **Capture phase**—The framework executes the capture phase from the application root to the source component until all components are traversed. Any handling event can stop propagation by calling `stopPropagation()` on the event.
3. **Bubble phase**—The framework executes the bubble phase from the source component to the application root until all components are traversed or `stopPropagation()` is called.

Note: Application events have a separate default phase. There's no separate default phase for component events. The default phase is the bubble phase.

Create Custom Component Events

Create a custom component event using the `<aura:event>` tag in a `.evt` resource. Events can contain attributes that can be set before the event is fired and read when the event is handled.

Use `type="COMPONENT"` in the `<aura:event>` tag for a component event. For example, this `c:compEvent` component event has one attribute with a name of `message`.

```
<!--c:compEvent-->
<aura:event type="COMPONENT">
    <!-- Add aura:attribute tags to define event shape.
         One sample attribute here. -->
    <aura:attribute name="message" type="String"/>
</aura:event>
```

The component that fires an event can set the event's data. To set the attribute values, call `event.setParam()` or `event.setParams()`. A parameter name set in the event must match the `name` attribute of an `<aura:attribute>` in the event. For example, if you fire `c:compEvent`, you could use:

```
event.setParam("message", "event message here");
```

The component that handles an event can retrieve the event data. To retrieve the attribute value in this event, call `event.getParam("message")` in the handler's client-side controller.

Fire Component Events

Fire a component event to communicate data to another component. A component event can be handled by the component that fired the event or by a component in the containment hierarchy that receives the event.

Register an Event

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

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

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

Fire an Event

To get a reference to a component event in JavaScript, use `getEvent("evtName")` where `evtName` matches the `name` attribute in `<aura:registerEvent>`.

Use `fire()` to fire the event from an instance of a component. For example, in an action function in a client-side controller:

```
var compEvent = cmp.getEvent("sampleComponentEvent");
// Optional: set some data for the event (also known as event shape)
// A parameter's name must match the name attribute
// of one of the event's <aura:attribute> tags
// compEvent.setParams({"myParam" : myValue });
compEvent.fire();
```

Handling Component Events

A component event can be handled by the component that fired the event or by a component in the containment hierarchy that receives the event.

Use `<aura:handler>` in the markup of the handler component. For example:

```
<aura:handler name="sampleComponentEvent" event="c:compEvent"
    action="{!c.handleComponentEvent}"/>
```

The `name` attribute in `<aura:handler>` must match the `name` attribute in the `<aura:registerEvent>` tag in the component that fires the event.

The `action` attribute of `<aura:handler>` sets the client-side controller action to handle the event.

The `event` attribute specifies the event being handled. The format is ***`namespace:eventName`***.

In this example, when the event is fired, the `handleComponentEvent` client-side controller action is called.

Event Handling Phases

Component event handlers are associated with the bubble phase by default. To add a handler for the capture phase instead, use the `phase` attribute.

```
<aura:handler name="sampleComponentEvent" event="ns:eventName"
    action="{!c.handleComponentEvent}" phase="capture" />
```

Get the Source of an Event

In the client-side controller action for an `<aura:handler>` tag, use `evt.getSource()` to find out which component fired the event, where `evt` is a reference to the event. To retrieve the source element, use `evt.getSource().getElement()`.

IN THIS SECTION:

Component Handling Its Own Event

A component can handle its own event by using the `<aura:handler>` tag in its markup.

Handling Bubbled or Captured Component Events

Event propagation rules determine which components in the containment hierarchy can handle events by default in the bubble or capture phases. Learn about the rules and how to handle events in the bubble or capture phases.

Handling Component Events Dynamically

A component can have its handler bound dynamically via JavaScript. This is useful if a component is created in JavaScript on the client-side.

SEE ALSO:

Component Event Propagation

Component Handling Its Own Event

A component can handle its own event by using the `<aura:handler>` tag in its markup.

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

```
<aura:registerEvent name="sampleComponentEvent" type="c:compEvent"/>
<aura:handler name="sampleComponentEvent" event="c:compEvent"
    action="{!c.handleSampleEvent}"/>
```

Note: The `name` attributes in `<aura:registerEvent>` and `<aura:handler>` must match, since each event is defined by its name.

Handling Bubbled or Captured Component Events

Event propagation rules determine which components in the containment hierarchy can handle events by default in the bubble or capture phases. Learn about the rules and how to handle events in the bubble or capture phases.

The framework supports *capture* and *bubble* phases for the propagation of component events. These phases are similar to DOM handling patterns and provide an opportunity for interested components to interact with an event and potentially control the behavior for subsequent handlers. The capture phase executes before the bubble phase.

Default Event Propagation Rules

By default, every parent in the containment hierarchy can't handle an event during the capture and bubble phases. Instead, the event propagates to every owner in the containment hierarchy.

A component's owner is the component that is responsible for its creation. For declaratively created components, the owner is the outermost component containing the markup that references the component firing the event. For programmatically created components, the owner component is the component that invoked `$A.createComponent` to create it.

The same rules apply for the capture phase, although the direction of event propagation (down) is the opposite of the bubble phase (up).

Confused? It makes more sense when you look at an example in the bubbling phase.

`c:owner` contains `c:container`, which in turn contains `c:eventSource`.

```
<!--c:owner-->
<aura:component>
    <c:container>
        <c:eventSource />
    </c:container>
</aura:component>
```

If `c:eventSource` fires an event, it can handle the event itself. The event then bubbles up the containment hierarchy.

`c:container` contains `c:eventSource` but it's not the owner because it's not the outermost component in the markup, so it can't handle the bubbled event.

`c:owner` is the owner because `c:container` is in its markup. `c:owner` can handle the event.

Propagation to All Container Components

The default behavior doesn't allow an event to be handled by every parent in the containment hierarchy. Some components contain other components but aren't the owner of those components. These components are known as container components. In the example, `c:container` is a container component because it's not the owner for `c:eventSource`. By default, `c:container` can't handle events fired by `c:eventSource`.

A container component has a facet attribute whose type is `Aura.Component[]`, such as the default `body` attribute. The container component includes those components in its definition using an expression, such as `{!v.body}`. The container component isn't the owner of the components rendered with that expression.

To allow a container component to handle the event, add `includeFacets="true"` to the `<aura:handler>` tag of the container component. For example, adding `includeFacets="true"` to the handler in the container component, `c:container`, enables it to handle the component event bubbled from `c:eventSource`.

```
<aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"
    includeFacets="true" />
```

Handle Bubbled Event

A component that fires a component event registers that it fires the event by using the `<aura:registerEvent>` tag.

```
<aura:component>
    <aura:registerEvent name="compEvent" type="c:compEvent" />
</aura:component>
```

A component handling the event in the bubble phase uses the `<aura:handler>` tag to assign a handling action in its client-side controller.

```
<aura:component>
    <aura:handler name="compEvent" event="c:compEvent" action="{!c.handleBubbling}"/>
</aura:component>
```

Note: The `name` attribute in `<aura:handler>` must match the `name` attribute in the `<aura:registerEvent>` tag in the component that fires the event.

Handle Captured Event

A component handling the event in the capture phase uses the `<aura:handler>` tag to assign a handling action in its client-side controller.

```
<aura:component>
    <aura:handler name="compEvent" event="c:compEvent" action="{!c.handleCapture}"
        phase="capture" />
</aura:component>
```

The default handling phase for component events is bubble if no `phase` attribute is set.

Stop Event Propagation

Use the `stopPropagation()` method in the `Event` object to stop the event propagating to other components.

Pausing Event Propagation for Asynchronous Code Execution

Use `event.pause()` to pause event handling and propagation until `event.resume()` is called. This flow-control mechanism is useful for any decision that depends on the response from the execution of asynchronous code. For example, you might make a decision about event propagation based on the response from an asynchronous call to native mobile code.

You can call `pause()` or `resume()` in the capture or bubble phases.

Event Bubbling Example

Let's look at an example so you can play around with it yourself.

```
<!--c:eventBubblingParent-->
<aura:component>
    <c:eventBubblingChild>
        <c:eventBubblingGrandchild />
    </c:eventBubblingChild>
</aura:component>
```

First, we define a simple component event.

```
<!--c:compEvent-->
<aura:event type="COMPONENT">
    <!--simple event with no attributes-->
</aura:event>
```

`c:eventBubblingEmitter` is the component that fires `c:compEvent`.

```
<!--c:eventBubblingEmitter-->
<aura:component>
    <aura:registerEvent name="bubblingEvent" type="c:compEvent" />
    <ui:button press="{!c.fireEvent}" label="Start Bubbling"/>
</aura:component>
```

Here's the controller for `c:eventBubblingEmitter`. When you press the button, it fires the `bubblingEvent` event registered in the markup.

```
/*eventBubblingEmitterController.js*/
{
    fireEvent : function(cmp) {
        var cmpEvent = cmp.getEvent("bubblingEvent");
        cmpEvent.fire();
    }
}
```

`c:eventBubblingGrandchild` contains `c:eventBubblingEmitter` and uses `<aura:handler>` to assign a handler for the event.

```
<!--c:eventBubblingGrandchild-->
<aura:component>
    <aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"/>


    <div class="grandchild">
        <c:eventBubblingEmitter />
    </div>
</aura:component>
```

Here's the controller for `c:eventBubblingGrandchild`.

```
/*eventBubblingGrandchildController.js*/
{
    handleBubbling : function(component, event) {
        console.log("Grandchild handler for " + event.getName());
    }
}
```

The controller logs the event name when the handler is called.

Here's the markup for `c:eventBubblingChild`. We will pass `c:eventBubblingGrandchild` in as the body of `c:eventBubblingChild` when we create `c:eventBubblingParent` later in this example.

```
<!--c:eventBubblingChild-->
<aura:component>
    <aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"/>


    <div class="child">
        {!v.body}
    </div>
</aura:component>
```

Here's the controller for `c:eventBubblingChild`.

```
/*eventBubblingChildController.js*/
{
    handleBubbling : function(component, event) {
        console.log("Child handler for " + event.getName());
    }
}
```

`c:eventBubblingParent` contains `c:eventBubblingChild`, which in turn contains `c:eventBubblingGrandchild`.

```
<!--c:eventBubblingParent-->
<aura:component>
    <aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"/>


    <div class="parent">
        <c:eventBubblingChild>
            <c:eventBubblingGrandchild />
        </c:eventBubblingChild>
    </div>
</aura:component>
```

Here's the controller for `c:eventBubblingParent`.

```
/*eventBubblingParentController.js*/
{
    handleBubbling : function(component, event) {
        console.log("Parent handler for " + event.getName());
    }
}
```

Now, let's see what happens when you run the code.

1. In your browser, navigate to `c:eventBubblingParent`.
2. Click the **Start Bubbling** button that is part of the markup in `c:eventBubblingEmitter`.

3. Note the output in your browser's console:

   ```
   Grandchild handler for bubblingEvent
   Parent handler for bubblingEvent
   ```

The `c:compEvent` event is bubbled to `c:eventBubblingGrandchild` and `c:eventBubblingParent` as they are owners in the containment hierarchy. The event is not handled by `c:eventBubblingChild` as `c:eventBubblingChild` is in the markup for `c:eventBubblingParent` but it's not an owner as it's not the outermost component in that markup.

Now, let's see how to stop event propagation. Edit the controller for `c:eventBubblingGrandchild` to stop propagation.

```
/*eventBubblingGrandchildController.js*/
{
    handleBubbling : function(component, event) {
        console.log("Grandchild handler for " + event.getName());
        event.stopPropagation();
    }
}
```

Now, navigate to `c:eventBubblingParent` and click the **Start Bubbling** button.

Note the output in your browser's console:

```
Grandchild handler for bubblingEvent
```

The event no longer bubbles up to the `c:eventBubblingParent` component.

SEE ALSO:

Component Event Propagation

Handling Component Events Dynamically

A component can have its handler bound dynamically via JavaScript. This is useful if a component is created in JavaScript on the client-side. For more information, see Dynamically Adding Event Handlers on page 169.

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.

Component Event

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

```
<!--c:ceEvent-->
<aura:event type="COMPONENT">
    <aura:attribute name="message" type="String"/>
</aura:event>
```

Notifier Component

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

The button in the component contains 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.

```
<!--c:ceNotifier-->
<aura:component>
    <aura:registerEvent name="cmpEvent" type="c:ceEvent"/>

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

The client-side controller gets an instance of the event by calling `cmp.getEvent("cmpEvent")`, where `cmpEvent` matches the value of the name attribute in the `<aura:registerEvent>` tag in the component markup. The controller sets the `message` attribute of the event and fires the event.

```
/* ceNotifierController.js */
{
    fireComponentEvent : function(cmp, event) {
        // Get the component event by using the
        // name value from aura:registerEvent
        var cmpEvent = cmp.getEvent("cmpEvent");
        cmpEvent.setParams({
            "message" : "A component event fired me. " +
            "It all happened so fast. Now, I'm here!" });
        cmpEvent.fire();
    }
}
```

Handler Component

The `c:ceHandler` handler component contains the `c:ceNotifier` component. The `<aura:handler>` tag uses the same value of the `name` attribute, `cmpEvent`, from the `<aura:registerEvent>` tag in `c:ceNotifier`. This wires up `c:ceHandler` to handle the event bubbled up from `c:ceNotifier`.

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

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

    <!-- Note that name="cmpEvent" in aura:registerEvent
     in ceNotifier.cmp -->
    <aura:handler name="cmpEvent" event="c:ceEvent" action="{!c.handleComponentEvent}"/>

    <!-- handler contains the notifier component -->
    <c:ceNotifier />
    <p>{!v.messageFromEvent}</p>
    <p>Number of events: {!v.numEvents}</p>

</aura:component>
```

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

```
/* ceHandlerController.js */
{
    handleComponentEvent : function(cmp, event) {
        var message = event.getParam("message");

        // set the handler attributes based on event data
        cmp.set("v.messageFromEvent", message);
        var numEventsHandled = parseInt(cmp.get("v.numEvents")) + 1;
        cmp.set("v.numEvents", numEventsHandled);
    }
}
```

Put It All Together

Navigate to the `c:ceHandler` component and click the button to fire the component event.

`http://localhost:<port>/c/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:

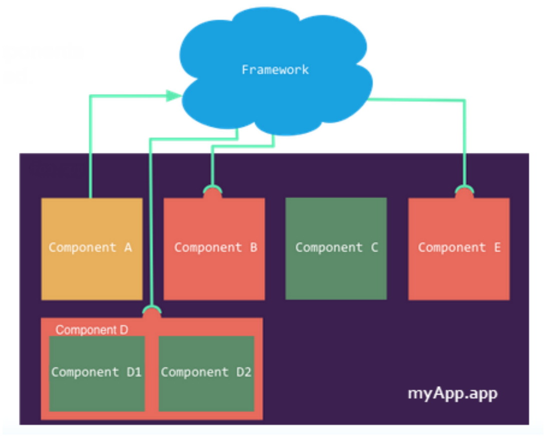
Component Events

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.



IN THIS SECTION:

SEE ALSO:

Component Events

Handling Events with Client-Side Controllers

Application Event Propagation

Advanced Events Example

Application Event Propagation

The framework supports *capture*, *bubble*, and *default* phases for the propagation of application events. The capture and bubble phases are similar to DOM handling patterns and provide an opportunity for interested components to interact with an event and potentially control the behavior for subsequent handlers. The default phase preserves the framework's original handling behavior.

The component that fires an event is known as the source component. The framework allows you to handle the event in different phases. These phases give you flexibility for how to best process the event for your application.

The phases are:

Capture

The event is captured and trickles down from the application root to the source component. The event can be handled by a component in the containment hierarchy that receives the captured event.

Event handlers are invoked in order from the application root down to the source component that fired the event.

Any registered handler in this phase can stop the event from propagating, at which point no more handlers are called in this phase or the bubble phase. If a component stops the event propagation using `event.stopPropagation()`, the component becomes the root node used in the default phase.

Any registered handler in this phase can cancel the default behavior of the event by calling `event.preventDefault()`. This call prevents execution of any of the handlers in the default phase.

Bubble

The component that fired the event can handle it. The event then bubbles up from the source component to the application root. The event can be handled by a component in the containment hierarchy that receives the bubbled event.

Event handlers are invoked in order from the source component that fired the event up to the application root.

Any registered handler in this phase can stop the event from propagating, at which point no more handlers will be called in this phase. If a component stops the event propagation using `event.stopPropagation()`, the component becomes the root node used in the default phase.

Any registered handler in this phase can cancel the default behavior of the event by calling `event.preventDefault()`. This call prevents execution of any of the handlers in the default phase.

Default

Event handlers are invoked in a non-deterministic order from the root node through its subtree. The default phase doesn't have the same propagation rules related to component hierarchy as the capture and bubble phases. The default phase can be useful for handling application events that affect components in different sub-trees of your app.

If the event's propagation wasn't stopped in a previous phase, the root node defaults to the application root. If the event's propagation was stopped in a previous phase, the root node is set to the component whose handler invoked `event.stopPropagation()`.

Here is the sequence of application event propagation.

1. **Event fired**—An application event is fired. The component that fires the event is known as the source component.
2. **Capture phase**—The framework executes the capture phase from the application root to the source component until all components are traversed. Any handling event can stop propagation by calling `stopPropagation()` on the event.
3. **Bubble phase**—The framework executes the bubble phase from the source component to the application root until all components are traversed or `stopPropagation()` is called.
4. **Default phase**—The framework executes the default phase from the root node unless `preventDefault()` was called in the capture or bubble phases. If the event's propagation wasn't stopped in a previous phase, the root node defaults to the application root. If the event's propagation was stopped in a previous phase, the root node is set to the component whose handler invoked `event.stopPropagation()`.

Create Custom Application Events

Create a custom application event using the `<aura:event>` tag in a `.evt` resource. Events can contain attributes that can be set before the event is fired and read when the event is handled.

Use `type="APPLICATION"` in the `<aura:event>` tag for an application event. For example, this `c:appEvent` application event has one attribute with a name of `message`.

```
<!--c:appEvent-->
<aura:event type="APPLICATION">
    <!-- Add aura:attribute tags to define event shape.
         One sample attribute here. -->
    <aura:attribute name="message" type="String"/>
</aura:event>
```

The component that fires an event can set the event's data. To set the attribute values, call `event.setParam()` or `event.setParams()`. A parameter name set in the event must match the `name` attribute of an `<aura:attribute>` in the event. For example, if you fire `c:appEvent`, you could use:

```
event.setParam("message", "event message here");
```

The component that handles an event can retrieve the event data. To retrieve the attribute in this event, call `event.getParam("message")` in the handler's client-side controller.

Fire Application Events

Application events follow a traditional publish-subscribe model. An application event is fired from an instance of a component. All components that provide a handler for the event are notified.

Register an Event

A component registers that it may fire an application event by using `<aura:registerEvent>` in its markup. The `name` attribute is required but not used for application events. The `name` attribute is only relevant for component events. This example uses `name="appEvent"` but the value isn't used anywhere.

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

Fire an Event

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

```
var appEvent = $A.get("e.c:appEvent");
// Optional: set some data for the event (also known as event shape)
// A parameter’s name must match the name attribute
// of one of the event’s <aura:attribute> tags
//appEvent.setParams({ "myParam" : myValue });
appEvent.fire();
```

Events Fired on App Rendering

Several events are fired when an app is rendering. All `init` events are fired to indicate the component or app has been initialized. If a component is contained in another component or app, 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. For more information, see Events Fired During the Rendering Lifecycle on page 122.

Handling Application Events

Use `<aura:handler>` in the markup of the handler component.

For example:

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

The `event` attribute specifies the event being handled. The format is ***`namespace:eventName`***.

The `action` attribute of `<aura:handler>` sets the client-side controller action to handle the event.

In this example, when the event is fired, the `handleApplicationEvent` client-side controller action is called.

Event Handling Phases

The framework allows you to handle the event in different phases. These phases give you flexibility for how to best process the event for your application.

Application event handlers are associated with the default phase. To add a handler for the capture or bubble phases instead, use the `phase` attribute.

Get the Source of an Event

In the client-side controller action for an `<aura:handler>` tag, use `evt.getSource()` to find out which component fired the event, where `evt` is a reference to the event. To retrieve the source element, use `evt.getSource().getElement()`.

IN THIS SECTION:

Handling Bubbled or Captured Application Events

Event propagation rules determine which components in the containment hierarchy can handle events by default in the bubble or capture phases. Learn about the rules and how to handle events in the bubble or capture phases.

Handling Bubbled or Captured Application Events

Event propagation rules determine which components in the containment hierarchy can handle events by default in the bubble or capture phases. Learn about the rules and how to handle events in the bubble or capture phases.

The framework supports *capture*, *bubble*, and *default* phases for the propagation of application events. The capture and bubble phases are similar to DOM handling patterns and provide an opportunity for interested components to interact with an event and potentially control the behavior for subsequent handlers. The default phase preserves the framework's original handling behavior.

Default Event Propagation Rules

By default, every parent in the containment hierarchy can't handle an event during the capture and bubble phases. Instead, the event propagates to every owner in the containment hierarchy.

A component's owner is the component that is responsible for its creation. For declaratively created components, the owner is the outermost component containing the markup that references the component firing the event. For programmatically created components, the owner component is the component that invoked `$A.createComponent` to create it.

The same rules apply for the capture phase, although the direction of event propagation (down) is the opposite of the bubble phase (up).

Confused? It makes more sense when you look at an example in the bubbling phase.

`c:owner` contains `c:container`, which in turn contains `c:eventSource`.

```
<!--c:owner-->
<aura:component>
    <c:container>
        <c:eventSource />
    </c:container>
</aura:component>
```

If `c:eventSource` fires an event, it can handle the event itself. The event then bubbles up the containment hierarchy.

`c:container` contains `c:eventSource` but it's not the owner because it's not the outermost component in the markup, so it can't handle the bubbled event.

`c:owner` is the owner because `c:container` is in its markup. `c:owner` can handle the event.

Propagation to All Container Components

The default behavior doesn't allow an event to be handled by every parent in the containment hierarchy. Some components contain other components but aren't the owner of those components. These components are known as container components. In the example, `c:container` is a container component because it's not the owner for `c:eventSource`. By default, `c:container` can't handle events fired by `c:eventSource`.

A container component has a facet attribute whose type is `Aura.Component[]`, such as the default `body` attribute. The container component includes those components in its definition using an expression, such as `{!v.body}`. The container component isn't the owner of the components rendered with that expression.

To allow a container component to handle the event, add `includeFacets="true"` to the `<aura:handler>` tag of the container component. For example, adding `includeFacets="true"` to the handler in the container component, `c:container`, enables it to handle the component event bubbled from `c:eventSource`.

```
<aura:handler name="bubblingEvent" event="c:compEvent" action="{!c.handleBubbling}"
    includeFacets="true" />
```

Handle Bubbled Event

To add a handler for the bubble phase, set `phase="bubble"`.

```
<aura:handler event="c:appEvent" action="{!c.handleBubbledEvent}"
    phase="bubble" />
```

The `event` attribute specifies the event being handled. The format is ***`namespace:eventName`***.

The `action` attribute of `<aura:handler>` sets the client-side controller action to handle the event.

Handle Captured Event

To add a handler for the capture phase, set `phase="capture"`.

```
<aura:handler event="c:appEvent" action="{!c.handleCapturedEvent}"
    phase="capture" />
```

Stop Event Propagation

Use the `stopPropagation()` method in the `Event` object to stop the event propagating to other components.

Pausing Event Propagation for Asynchronous Code Execution

Use `event.pause()` to pause event handling and propagation until `event.resume()` is called. This flow-control mechanism is useful for any decision that depends on the response from the execution of asynchronous code. For example, you might make a decision about event propagation based on the response from an asynchronous call to native mobile code.

You can call `pause()` or `resume()` in the capture or bubble phases.

Application Event Example

Here's a simple use case of using an application event to update an attribute in another component.

1. A user clicks a button in the notifier component, `aeNotifier.cmp`.
2. The client-side controller for `aeNotifier.cmp` sets a message in a component event and fires the event.

3. The handler component, `aeHandler.cmp`, handles the fired event.
4. The client-side controller for `aeHandler.cmp` sets an attribute in `aeHandler.cmp` based on the data sent in the event.

Application Event

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

```
<!--c:aeEvent-->
<aura:event type="APPLICATION">
    <aura:attribute name="message" type="String"/>
</aura:event>
```

Notifier Component

The `aeNotifier.cmp` notifier component uses `aura:registerEvent` to declare that it may fire the application event. The `name` attribute is required but not used for application events. The `name` attribute is only relevant for component events.

The button in the component contains a `press` browser event that is wired to the `fireApplicationEvent` action in the client-side controller. Clicking this button invokes the action.

```
<!--c:aeNotifier-->
<aura:component>
    <aura:registerEvent name="appEvent" type="c:aeEvent"/>

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

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

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

Handler Component

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

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

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

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

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

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

```
/* aeHandlerController.js */
{
    handleApplicationEvent : function(cmp, event) {
        var message = event.getParam("message");

        // set the handler attributes based on event data
        cmp.set("v.messageFromEvent", message);
        var numEventsHandled = parseInt(cmp.get("v.numEvents")) + 1;
        cmp.set("v.numEvents", numEventsHandled);
    }
}
```

Container Component

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

```
<!--c:aeContainer-->
<aura:component>
    <c:aeNotifier/>
    <c:aeHandler/>
</aura:component>
```

Put It All Together

You can test this code by navigating to:

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

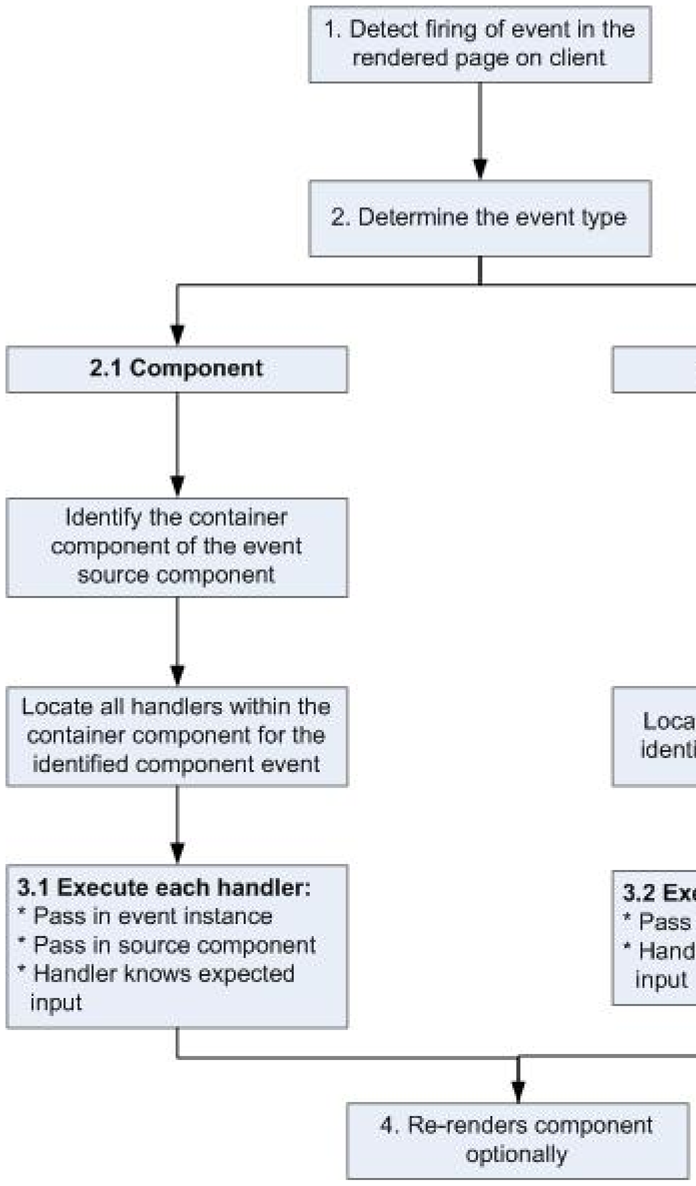
If you want to access data on the server, you could extend this example to call a server-side controller from the handler's client-side controller.

SEE ALSO:

- Application Events
- Creating Server-Side Logic with Controllers
- Component Event Example

Event Handling Lifecycle

The following chart summarizes how the framework handles events.


1. Detect firing of event in the rendered page on client
2. Determine the event type
2.1 Component
2.2 Application
Identify the container component of the event source component
Locate all handlers within the container component for the identified component event
Locate all handlers for the identified application event
3.1 Execute each handler:
* Pass in event instance
* Pass in source component
* Handler knows expected input
3.2 Execute each handler:
* Pass in event instance
* Handler knows expected input
4. Re-renders component optionally

1 Detect Firing of Event

The framework detects the firing of an event. For example, the event could be triggered by a button click in a notifier component.

2 Determine the Event Type

2.1 Component Event

The parent or container component instance that fired the event is identified. This container component locates all relevant event handlers for further processing.

2.2 Application Event

Any component can have an event handler for this event. All relevant event handlers are located.

3 Execute each Handler

3.1 Executing a Component Event Handler

Each of the event handlers defined in the container component for the event are executed by the handler controller, which can also:

- Set attributes or modify data on the component (causing a re-rendering of the component).
- Fire another event or invoke a client-side or server-side action.

3.2 Executing an Application Event Handler

All event handlers are executed. When the event handler is executed, the event instance is passed into the event handler.

4 Re-render Component (optional)

After the event handlers and any callback actions are executed, a component might be automatically re-rendered if it was modified during the event handling process.

SEE ALSO:

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

The definitions of component and application events are stored in separate `.evt` resources, but individual notifier and handler component bundles can contain code to work with both types of events.

The component and application events both contain a `context` attribute that defines the shape of the event. This is the data that is passed to handlers of the event.

Component Event

Here is the markup for `compEvent.evt`.

```
<!--c:compEvent-->
<aura:event type="COMPONENT">
    <!-- pass context of where the event was fired to the handler. -->
    <aura:attribute name="context" type="String"/>
</aura:event>
```

Application Event

Here is the markup for `appEvent.evt`.

```
<!--c:appEvent-->
<aura:event type="APPLICATION">
    <!-- pass context of where the event was fired to the handler. -->
    <aura:attribute name="context" type="String"/>
</aura:event>
```

Notifier Component

The `eventsNotifier.cmp` notifier component contains buttons to initiate a component or application event.

The notifier uses `aura:registerEvent` tags to declare that it may fire the component and application events. Note that the `name` attribute is required but the value is only relevant for the component event; the value is not used anywhere else for the application event.

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

```
<!--c:eventsNotifier-->
<aura:component>
  <aura:attribute name="parentName" type="String"/>
  <aura:registerEvent name="componentEventFired" type="c:compEvent"/>
  <aura:registerEvent name="appEvent" type="c:appEvent"/>

  <div>
    <h3>This is {!v.parentName}'s eventsNotifier.cmp instance</h3>
    <p><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

The CSS is in `eventsNotifier.css`.

```
/* eventsNotifier.css */
.cEventsNotifier {
    display: block;
    margin: 10px;
    padding: 10px;
    border: 1px solid black;
}
```

Client-side controller source

The `eventsNotifierController.js` controller fires the event.

```
/* eventsNotifierController.js */
{
    fireComponentEvent : function(cmp, event) {
        var parentName = cmp.get("v.parentName");

        // Look up event by name, not by type
        var compEvents = cmp.getEvent("componentEventFired");

        compEvents.setParams({ "context" : parentName });
        compEvents.fire();
    },

    fireApplicationEvent : function(cmp, event) {
        var parentName = cmp.get("v.parentName");

        // note different syntax for getting application event
        var appEvent = $A.get("e.c:appEvent");

        appEvent.setParams({ "context" : parentName });
        appEvent.fire();
    }
}
```

You can click the buttons to fire component and application events but there is no change to the output because we haven't wired up the handler component to react to the events yet.

The controller sets the `context` attribute of the component or application event to the `parentName` of the notifier component before firing the event. We will see how this affects the output when we look at the handler component.

Handler Component

The `eventsHandler.cmp` handler component contains the `c:eventsNotifier` notifier component and `<aura:handler>` tags for the application and component events.

```
<!--c:eventsHandler-->
<aura:component>
  <aura:attribute name="name" type="String"/>
  <aura:attribute name="mostRecentEvent" type="String" default="Most recent event handled:"/>

  <aura:attribute name="numComponentEventsHandled" type="Integer" default="0"/>
  <aura:attribute name="numApplicationEventsHandled" type="Integer" default="0"/>

  <aura:handler event="c:appEvent" action="{!c.handleApplicationEventFired}"/>
  <aura:handler name="componentEventFired" event="c:compEvent"
action="{!c.handleComponentEventFired}"/>
```

```
  <div>
    <h3>This is {!v.name}</h3>
    <p>{!v.mostRecentEvent}</p>
    <p># component events handled: {!v.numComponentEventsHandled}</p>
    <p># application events handled: {!v.numApplicationEventsHandled}</p>
    <c:eventsNotifier parentName="{#v.name}" />
  </div>
</aura:component>
```

Note: `{#v.name}` is an unbound expression. This means that any change to the value of the `parentName` attribute in `c:eventsNotifier` doesn't propagate back to affect the value of the `name` attribute in `c:eventsHandler`. For more information, see Data Binding Between Components on page 27.

CSS source

The CSS is in `eventsHandler.css`.

```
/* eventsHandler.css */
.cEventsHandler {
  display: block;
  margin: 10px;
  padding: 10px;
  border: 1px solid black;
}
```

Client-side controller source

The client-side controller is in `eventsHandlerController.js`.

```
/* eventsHandlerController.js */
{
    handleComponentEventFired : function(cmp, event) {
        var context = event.getParam("context");
        cmp.set("v.mostRecentEvent",
            "Most recent event handled: COMPONENT event, from " + context);

        var numComponentEventsHandled =
            parseInt(cmp.get("v.numComponentEventsHandled")) + 1;
        cmp.set("v.numComponentEventsHandled", numComponentEventsHandled);
    },

    handleApplicationEventFired : function(cmp, event) {
        var context = event.getParam("context");
        cmp.set("v.mostRecentEvent",
            "Most recent event handled: APPLICATION event, from " + context);

        var numApplicationEventsHandled =
            parseInt(cmp.get("v.numApplicationEventsHandled")) + 1;
        cmp.set("v.numApplicationEventsHandled", numApplicationEventsHandled);
    }
}
```

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

You can click buttons and the UI now changes to indicate the type of event. The click count increments to indicate whether it's a component or application event. We aren't finished yet though. Notice that the source of the event is undefined as the event `context` attribute hasn't been set .

Container Component

Here is the markup for `eventsContainer.cmp`.

```
<!--c:eventsContainer-->
<aura:component>
    <c:eventsHandler name="eventsHandler1"/>
    <c:eventsHandler name="eventsHandler2"/>
</aura:component>
```

The container component contains two handler components. It sets the `name` attribute of both handler components, which is passed through to set the `parentName` attribute of the notifier components. This fills in the gaps in the UI text that we saw when we looked at the notifier or handler components directly.

Navigate to the `c:eventsContainer` component.

`http://localhost:<port>/c/eventsContainer.cmp`.

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 Aura events from JavaScript code outside an Aura app. For example, your Aura app might need to call out to some non-Aura code, and then have that code communicate back to your Aura app once it's done.

For example, you could call external code that needs to log into another system and return some data to your Aura app. 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 Outside the Framework Lifecycle

Events Best Practices

Here are some best practices for working with events.

Use Component Events Whenever Possible

Always try to use a component event instead of an application event, if possible. Component events can only be handled by components above them in the containment hierarchy so their usage is more localized to the components that need to know about them. Application events are best used for something that should be handled at the application level, such as navigating to a specific record. Application events allow communication between components that are in separate parts of the application and have no direct containment relationship.

Separate Low-Level Events from Business Logic Events

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 155.

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

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

SEE ALSO:

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.

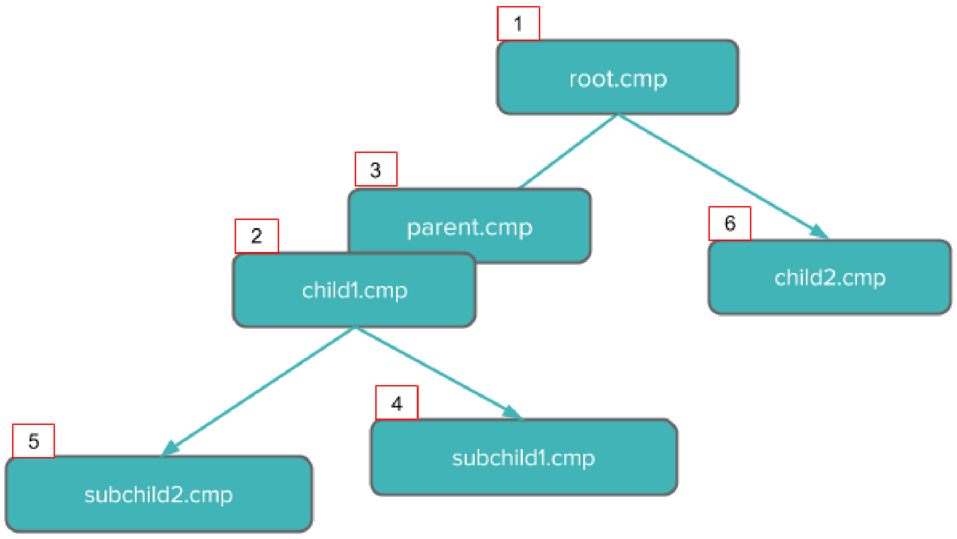
Component Creation

The component lifecycle starts when the client sends an HTTP request to the server and the component configuration data is returned to the client. No server trip is made if the component definition is already on the client from a previous request and the component has no server dependencies.

Let's look at an app with several nested components. The framework instantiates the app and goes through the children of the `v.body` facet to create each component, First, it creates the component definition, its entire parent hierarchy, and then creates the facets within those components. The framework also creates any component dependencies on the server, including definitions for attributes, interfaces, controllers, actions, and models.

For an abstract component, your JavaScript or Java provider determines which concrete implementation of the component to create.

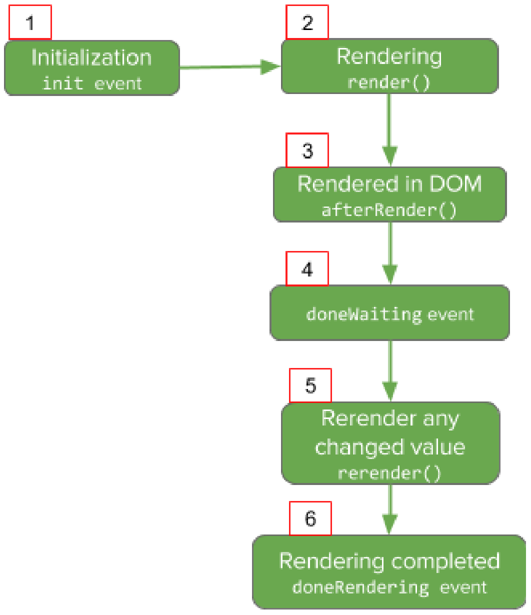
The following image lists the order of component creation.



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 that's used to render the component instance. When the component tree is ready, the `init` event is fired for all the components, starting from the children component and finishing in the parent component.

Component Rendering

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 before the component starts rendering. For more information, see Invoking Actions on Component Initialization on page 155.

2. For each component in the tree, the base implementation of `render()` or your custom renderer is called to start component rendering. For more information, see Client-Side Rendering to the DOM on page 150. Similar to the component creation process, rendering starts at the root component, its children components and their super components, if any, and finally the subchildren components.

3. Once your components are rendered to the DOM, `afterRender()` is called to signal that rendering is completed for each of these component definitions. It enables you to interact with the DOM tree after the framework rendering service has created the DOM elements.

4. To indicate that the client is done waiting for a response to the server request XHR, the `doneWaiting` event is fired. You can handle this event by adding a handler wired to a client-side controller action.

5. The framework checks whether any components need to be rerendered and rerenders any "dirtied" components to reflect any updates to attribute values. This rerender check is done even if there's no dirtied components or values.

6. Finally, the `doneRendering` event is fired at 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
`uiExamples:buttonExample`	The top-level component that contains the `ui:button` component
`ui:button`	The `ui:button` component that's in the top-level component
`aura:html`	Renders the `<button>` tag.
`aura:if`	The first `aura:if` tag in `ui:button`, which doesn't render anything since the button contains no image

Component	Description
`aura:if`	The second `aura:if` tag in `ui:button`
`aura:html`	The `<span>` tag for the button label, nested in the `<button>` tag
`aura:expression`	The `v.num` expression
`aura:expression`	Empty `v.body` expression

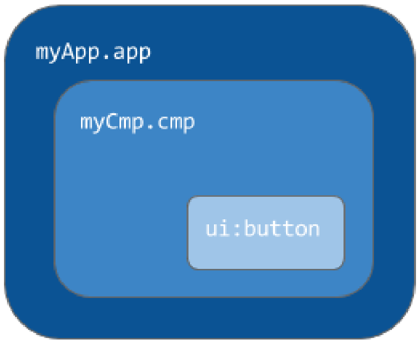
HTML tags in the markup are converted to `aura:html`, which has a `tag` attribute that defines the HTML tag to be generated. When rendering is done, this example calls `afterRender()` eight times for these component definitions. The `doneWaiting` event is fired, followed by the `doneRendering` event.

Clicking the button updates its label, which checks for any "dirtied" 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 "dirtied" 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 `myCmp.cmp` with a `ui:button` component.



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

SEE ALSO:

- Client-Side Rendering to the DOM
- Server-Side Processing for Component Requests
- Client-Side Processing for Component Requests
- System Event Reference

System Events

The framework fires several system events during its lifecycle.

You can handle these events in your Lightning apps or components, and within Salesforce1.

Event Name	Description
`aura:doneRendering`	Indicates that the initial rendering of the root application or root component has completed.
`aura:doneWaiting`	Indicates that the app or component is done waiting for a response to a server request. This event is preceded by an `aura:waiting` event.
`aura:locationChange`	Indicates that the hash part of the URL has changed.
`aura:noAccess`	Indicates that a requested resource is not accessible due to security constraints on that resource.
`aura:systemError`	Indicates that an error has occurred.
`aura:valueChange`	Indicates that an attribute value has changed.
`aura:valueDestroy`	Indicates that a component has been destroyed.
`aura:valueInit`	Indicates that an app or component has been initialized.
`aura:waiting`	Indicates that the app or component is waiting for a response to a server request.

SEE ALSO:

System Event Reference

CHAPTER 5 Creating Apps

In this chapter ...

Components are the building blocks of an app. This section shows you a typical workflow to put the pieces together to create a new app.

App Overview

An app is a special top-level component whose markup is in a `.app` file.

On a production server, the `.app` file is the only addressable unit in a browser URL. Access an app using the URL:

`http://<myServer>/<namespace>/<appName>.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. Each part of your UI corresponds to a component, which can in turn contain nested components. Compose components to create a sophisticated app.

An app's markup starts with the `<aura:application>` tag.

To learn more about the `<aura:application>` tag, see aura:application.

Let's look at a `sample.app` file, which starts with the `<aura:application>` tag.

```
<aura:application>
    <div>
      <header>
          <h1>Sample App</h1>
      </header>
      <ui:block class="wrapper" aura:id="block">
        <aura:set attribute="left">
             <docsample:sidebar aura:id="sidebar" />
        </aura:set>
             <docsample:details aura:id="details" />
      </ui:block>
    </div>
</aura:application>
```

The `sample.app` file 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 `<docsample:sidebar>` and `<docsample:details>` components encapsulate the layout for the page.

SEE ALSO:

aura:application

Creating App Templates

An app template bootstraps the loading of the framework and the app. Customize an app's template by creating a component that extends the default `aura:template` template.

A template must have the `isTemplate` system attribute in the `<aura:component>` tag set to `true`. This informs the framework to allow restricted items, such as `<script>` tags, which aren't allowed in regular components.

For example, a sample app has a `np:template` template that extends `aura:template`. `np:template` looks like:

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

Note how the component extends `aura:template` and sets the `title` attribute using `aura:set`.

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

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

A template can only extend a component or another template. A component or an application can't extend a template.

JavaScript Libraries

You can reference a JavaScript library in your app's template. For other options, see Using External JavaScript Libraries on page 143.

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.

This sample template markup references a third-party JavaScript library.

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

CSS in Components

Using External JavaScript Libraries

Developing Secure Code

The LockerService architectural layer enhances security by isolating individual Lightning components in their own containers and enforcing coding best practices.

IN THIS SECTION:

Writing Secure Code with LockerService

LockerService is a powerful security architecture for Lightning components. LockerService enhances security by isolating individual Lightning components in their own containers. LockerService also promotes best practices that improve the supportability of your code by only allowing access to supported APIs and eliminating access to non-published framework internals.

LockerService Requirements

LockerService enforces several security features in your code.

JavaScript ES5 Strict Mode Enforcement

JavaScript ES5 strict mode is implicitly enabled. You don't need to specify `"use strict"` in your code. Enforcement includes declaration of variables with the `var` keyword and other JavaScript coding best practices. The libraries that your components use must also work in strict mode.

DOM Access Containment

A component can only traverse the DOM and access elements created by a component in the same namespace. This behavior prevents the anti-pattern of reaching into DOM elements owned by components in another namespace.

Note: It's an anti-pattern for any component to "reach into" another component, regardless of namespace. LockerService only prevents cross-namespace access. Your good judgment should prevent cross-component access within your own namespace.

Restrictions to Global References

LockerService applies restrictions to global references. You can access intrinsic objects, such as `Array`. LockerService provides secure versions of non-intrinsic objects, such as `window`. The secure object versions automatically and seamlessly control access to the object and its properties.

Access to Supported JavaScript API Framework Methods Only

You can access published, supported JavaScript API framework methods only. These methods are published in the reference doc app at `https://yourDomain.lightning.force.com/auradocs/reference.app`. Previously, unsupported methods were accessible, which exposed your code to the risk of breaking when unsupported methods were changed or removed.

Don't Use `instanceof`

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

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`.

Besides CSS that styles the component in the bundle, you can add a flavor CSS file using the file name `<componentName>Flavors.css`. Flavors enable you to style different component instances easily and apply various styles to components within a namespace.

IN THIS SECTION:

Using External CSS

To use external CSS in your app, add it to your app's template.

More Readable Styling Markup with the join Expression

Markup can get messy when you specify the class names to apply based on the component attribute values. Try using a `join` expression for easier-to-read markup.

Styling with Flavors

A flavor provides stylistic variations of a component. Each flavor is essentially a CSS class that can apply varying styles to different instances of a components.

Styling with Tokens
Tokens make it easy to ensure that your design is consistent, and even easier to update it as your design evolves. Define the token values once and reuse them throughout your Aura applications.

SEE ALSO:
CSS in Components
Creating App Templates
Add Lightning Components as Custom Tabs in Salesforce1

Using External CSS

To use external CSS in your app, add it to your app's template.

SEE ALSO:
Using External JavaScript Libraries

More Readable Styling Markup with the `join` Expression

Markup can get messy when you specify the class names to apply based on the component attribute values. Try using a `join` expression for easier-to-read markup.

This example sets the class names based on the component attribute values. It's readable, but the spaces between class names are easy to forget.

```
<li class="{! 'calendarEvent ' +
    v.zoomDirection + ' ' +
    (v.past ? 'pastEvent ' : '') +
    (v.zoomed ? 'zoom ' : '') +
    (v.multiDayFragment ? 'multiDayFragment ' : '')}">
    <!-- content here -->
</li>
```

Sometimes, if the markup is not broken into multiple lines, it can hurt your eyes or make you mutter profanities under your breath.

```
<li class="{! 'calendarEvent ' + v.zoomDirection + ' ' + (v.past ? 'pastEvent ' : '') +
(v.zoomed ? 'zoom ' : '') + (v.multiDayFragment ? 'multiDayFragment ' : '')}">
    <!-- content here -->
</li>
```

Try using a `join` expression instead for easier-to-read markup. This example `join` expression sets `' '` as the first argument so that you don't have to specify it for each subsequent argument in the expression.

```
<li
    class="{! join(' ',
        'calendarEvent',
        v.zoomDirection,
        v.past ? 'pastEvent' : '',
        v.zoomed ? 'zoom' : '',
        v.multiDayFragment ? 'multiDayFragment' : ''
    )}">
    <!-- content here -->
</li>
```

You can also use a `join` expression for dynamic styling.

```
<div style="{! join(';',
    'top:' + v.timeOffsetTop + '%',
    'left:' + v.timeOffsetLeft + '%',
    'width:' + v.timeOffsetWidth + '%'
)}">
    <!-- content here -->
</div>
```

SEE ALSO:

Expression Functions Reference

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 the framework automatically adds any prefixes that are necessary when generating the CSS output. If you choose to add them, they are used as-is. This enables you to specify alternative values for certain prefixes.

Example: For example, this is an unprefixed version of `border-radius`.

```
.class {
  border-radius: 2px;
}
```

The previous declaration results in the following declarations.

```
.class {
  -webkit-border-radius: 2px;
  -moz-border-radius: 2px;
  border-radius: 2px;
}
```

Styling with Flavors

A flavor provides stylistic variations of a component. Each flavor is essentially a CSS class that can apply varying styles to different instances of a components.

Styling with flavors enables you to restyle the original components easily, but also avoid overriding any previous styling. When creating flavors, create them at the component level in component bundles.

To use flavors, you must make a component flavorable. Specify `aura:flavorable="true"` on the HTML element that should receive the flavor class name:

```
<aura:component>
  <div aura:flavorable="true">
     //other markup here
  </div>
</aura:component>
```

Only one element is marked flavorable, and it doesn't have to be a top-level element. For an example, see the `ui:button` component.

Creating and Setting a Flavor

You can create one or more flavors in a component bundle. Use the file name `<componentName>Flavors.css` to wire up the flavors to your component. Note that this CSS file for flavors is different than your regular `<componentName>.css` file. In the flavor CSS file, create your flavor in the format `.THIS--flavorName`. Each selector must be scoped by the flavor name class selector.

```
.THIS--info, .THIS--warning {
  border: 1px solid #eee;
  margin: t(margin);
}

.THIS--info { background-color: blue }
.THIS--warning { background-color: yellow }
```

When creating flavors for a component, move the existing CSS into the default flavor, if possible. The default flavor should be referenced using `.THIS`, which automatically converts to `.THIS--default`. For an example, see the `ui:button` component.

Note: Flavors are applied to an element if it matches the designated or default flavor of the component. When using flavors, we recommend applying little or no CSS in the regular CSS file, and using at least a default flavor in the flavor CSS file.

You can specify different flavor names. For example:

```
<!-- myComponent.cmp -->
<aura:component defaultFlavor="alternative">
  <div aura:flavorable="true">
    //other markup here
  </div>
</aura:component>

/* myComponentFlavors.css */
.THIS { color: red }
.THIS--alternative { color: green }
```

Setting a Flavor on Component Instances

When multiple flavors are available for a component, you can specify which one to use. For example:

```
<ui:button label="Search"/>
<ui:button aura:flavor="primary" label="Submit"/>
```

Specified flavors are not inherited or passed down to children.

You can specify multiple flavors on the same component instance by providing a comma-separated list of flavors in the `aura:flavor` attribute.

```
<ui:button aura:flavor="default, brand" label="Submit"/>
```

Make sure the flavors you specify work together for the component.

IN THIS SECTION:

SEE ALSO:

Dynamically Creating Components

Applying Flavors to a Namespace

You can add flavors for other components within your namespace.

Add flavors for components in another namespace in the flavors bundle (i.e., folder) within your own namespace. Create one flavored CSS file per component you want to flavor, in the format of `namespace-componentNameFlavors.css`, where namespace is the namespace of the component being flavored. For example, to create flavors for the `ui:button` component, create the CSS file `components/myNamespace/flavors/ui-buttonFlavors.css`.

```
<!-- ui/button/button.cmp -->
<aura:component>
  <button aura:flavorable="true">{!v.body}</button>
</aura:component>

/* ui/button/buttonFlavors.css */
.THIS {
  color: red;
  text-shadow: red 0 -2px;
}

/* sample/flavors/ui-buttonFlavors.css */
.THIS { color: green }
.THIS--special { font-weight: bold }
```

To apply the default and `special` flavors in the previous example, add a `.flavors` file to an application bundle. The next example adds flavors specified in `ui-buttonFlcavors.css` to all `ui:button` component instances in an application.

```
<!-- myApp.flavors -->
<aura:flavors>
  <aura:include source="sample:flavors"/>
</aura:flavors>
```

In this case, the default flavor of button will now be green, without affecting the `text-shadow` rule in the original flavor. By adding the `.flavors` file to the application bundle, the `special` flavor is now available for the component.

Styling with Tokens

Tokens make it easy to ensure that your design is consistent, and even easier to update it as your design evolves. Define the token values once and reuse them throughout your Aura applications.

There are two types of tokens: design tokens and configuration tokens.

Design Tokens
Design tokens are visual design “atoms” for building a design for your components or apps. Specifically, they’re named entities that store visual design attributes, such as pixel values for margins and spacing, font sizes and families, or hex values for colors. Use design tokens in CSS.

Capture the essential values of your visual design into named tokens. Tokens are a terrific way to centralize the low-level values, which you then use to compose the styles that make up the design of your component or app.

Configuration Tokens

Use configuration tokens in expressions, such as in component markup, for cross-cutting application values that remain consistent across all components in your app. A typical use case for configuration tokens is in a markup expression setting a class name.

It's a best practice to separate design tokens and configuration tokens into separate tokens bundles. However, the separation isn't enforced so you have freedom to create a token structure that works for you.

IN THIS SECTION:

Tokens Bundles
Tokens are a type of bundle, just like components, events, and interfaces. A tokens bundle only contains a tokens file.

Defining Tokens
A token is a name-value pair that you specify using the `<aura:token>` tag. A tokens file contains one or more tokens.

Using Design Tokens in CSS
Use tokens in your component's CSS for consistent styling across all components in your app.

Using Configuration Tokens in Expressions
Use configuration tokens in expressions, such as in component markup, for cross-cutting application values that remain consistent across all components in your app. A typical use case for configuration tokens is in a markup expression setting a class name.

Tokens Bundles

Tokens are a type of bundle, just like components, events, and interfaces. A tokens bundle only contains a tokens file.

CSS Tokens Bundle for a Namespace

The tokens file for a namespace is automatically loaded for usage in CSS if it follows this file naming convention:

`<myNamespace>/<myNamespace>Namespace/<myNamespace>Namespace.tokens`

That's a little confusing. Let's look at an example with the `docsample` namespace. The CSS tokens file is located at:

`aura-components/components/docsample/docsampleNamespace/docsampleNamespace.tokens`

If this file exists, the tokens can be used in any CSS files in the `docsample` namespace.

Custom Tokens Bundles

You can have multiple token bundles and you can name them whatever you want. You must manually wire up your custom token files.

You might want a custom tokens bundle for an application so that you can use tokens in markup or other expressions. For example:

`aura-components/components/docsample/myApp/myApp.tokens`

Wire up the custom tokens file in your application by using the `tokens` attribute in your `<aura:application>` tag:

```
<aura:application tokens="docsample:myApp">
    …
</aura:application>
```

Use comma-separated values for multiple tokens files. The naming convention for automatically loading tokens for a namespace is required for tokens in CSS, but can be augmented or overridden by using explicit token files. To use tokens in markup, you must explicitly set the tokens file in the `tokens` attribute. For example:

```
<aura:application tokens="docsample:myApp,docsample:docsampleNamespace">
    …
</aura:application>
```

Defining Tokens

A token is a name-value pair that you specify using the `<aura:token>` tag. A tokens file contains one or more tokens.

A tokens file starts with the `<aura:tokens>` tag. It can only contain `<aura:token>` tags to define tokens. The only allowed attributes for the `<aura:token>` tag are `name` and `value`.

For example:

```
<aura:tokens>
    <aura:token name="myBodyTextFontFace"
        value="'Helvetica, Arial, sans-serif"/>
    <aura:token name="myBodyTextFontWeight" value="normal"/>
    <aura:token name="myBackgroundColor" value="#f4f6f9"/>
    <aura:token name="myDefaultMargin" value="6px"/>
</aura:tokens>
```

IN THIS SECTION:

Using Expressions in Tokens Bundles

Tokens support a restricted set of expressions. Use expressions to reuse one token value in another token, or to combine tokens to form a more complex style property.

Using Expressions in Tokens Bundles

Tokens support a restricted set of expressions. Use expressions to reuse one token value in another token, or to combine tokens to form a more complex style property.

Cross-Referencing Tokens

To reference one token's value in another token's definition, wrap the token to be referenced in standard expression syntax.

In the following example, we look at two tokens files. `baseApp.tokens` defines some tokens. `extendApp.tokens` extends `baseApp.tokens` and uses expressions to reference the tokens from `baseApp.tokens`.

```
<!-- baseApp.tokens -->
<aura:tokens>
    ...
    <aura:token name="colorBackground" value="rgb(244, 246, 249)" />
    <aura:token name="fontFamily" value="'Salesforce Sans', Arial, sans-serif" />
  ...
</aura:tokens>
```

You can reference the tokens from `baseApp.tokens` in `extendApp.tokens`.

```
<!-- extendApp.tokens -->
<aura:tokens extends="docsample:baseApp">
    <aura:token name="mainColor" value="{! colorBackground }" />
    <aura:token name="btnColor" value="{! mainColor }" />
    <aura:token name="myFont" value="{! fontFamily }" />
</aura:tokens>
```

The `mainColor` token in `extendApp.tokens` uses an expression to reference the `colorBackground` token in `baseApp.tokens`.

You can only cross-reference tokens defined in the same file or a file you're extending.

Expression syntax in tokens files is restricted to references to other tokens.

Combining Tokens

To support combining individual token values into more complex CSS style properties, the `token()` function supports string concatenation. For example, if you have the following tokens defined:

```
<!-- myApp.tokens -->
<aura:tokens>
    <aura:token name="defaultHorizonalSpacing" value="12px" />
    <aura:token name="defaultVerticalSpacing" value="6px" />
</aura:tokens>
```

You can combine these two tokens in a CSS style definition. For example:

```
/* myComponent.css */
.THIS div.notification {
  margin: token(defaultVerticalSpacing + ' ' + defaultHorizonalSpacing);
  /* more styles here */
}
```

You can mix tokens with strings as much as necessary to create the right style definition. For example, use `margin: token(defaultVerticalSpacing + ' ' + defaultHorizonalSpacing + ' 3px');` to hard code the bottom spacing in the preceding definition.

The only operator supported within the `token()` function is "+" for string concatenation.

SEE ALSO:

Using Expressions

Using Design Tokens in CSS

Use tokens in your component's CSS for consistent styling across all components in your app.

To use a design token in a CSS file, reference it using the `token(`***`tokenName`***`)` function. For example:

```
.THIS p {
    font-family: token(myBodyTextFontFace);
    font-weight: token(myBodyTextFontWeight);
}
```

`myBodyTextFontFace` corresponds to the `name` attribute in an `<aura:token>` definition. The `token(myBodyTextFontFace)` function is replaced by the value in the `<aura:token>` definition.

If you prefer a more concise function name for referencing tokens, you can use the `t()` function instead of `token()`. The two are equivalent.

SEE ALSO:

CSS in Components

Using Configuration Tokens in Expressions

Use configuration tokens in expressions, such as in component markup, for cross-cutting application values that remain consistent across all components in your app. A typical use case for configuration tokens is in a markup expression setting a class name.

To use a token in markup, reference it using the `token('`***`tokenName`***`')` function. Note the single quotes around the token name. This syntax differs from CSS usage where there are no single quotes around the token name.

For example:

```
<ui:label class="{!token('labelClass')}" />
```

`labelClass` corresponds to the `name` attribute in an `<aura:token>` definition.

Note: To use a token in an expression:

- The `<aura:tokens>` tag in the tokens bundle containing the token must set the attribute `serialize="true"`. This isn't necessary for token usage in CSS as the CSS is parsed on the server before being serialized to the client. For example:

  ```
  <aura:tokens serialize="true">
      <aura:token name="labelClass"  value="'sampleClass"/>
  </aura:tokens>
  ```

- The `<aura:application>` tag must use the `tokens` attribute to point to the tokens bundle. For example:

  ```
  <aura:application tokens="docsample:docsampleNamespace">
  ```

Let's look at a complete sample. The `docsample/docsampleNamespace/docsampleNamespace.tokens` file defines a few tokens.

```
<aura:tokens serialize="true">
    <aura:token name="myBackgroundColor" value="#ff0000"/>
    <aura:token name="other" value="notFirst"/>
</aura:tokens>
```

We'll see how these tokens are used soon in CSS and markup. Notice that the `<aura:tokens>` tag sets `serialize="true"`.

The `docsample:tokensUsage` component includes the `{!token('other')}` tag to reference the `other` token.

```
<!--docsample:tokensUsage-->
<aura:component>
    <div class="white">
    Hello, HTML!
    </div>
    <h2>Check out the style in this list.</h2>
    <ul>
        <li class="first">I'm first.</li>
```

```
        <li class="{!token('other')}">I'm second.</li>
        <!--<li class="notFirst">I'm second.</li>-->
        <li>I'm third.</li>
    </ul>
</aura:component>
```

`<li class="{!token('other')}">I'm second.</li>` uses a token instead of the hard-coded equivalent that is commented out:

```
<!--<li class="notFirst">I'm second.</li>-->
```

Here is the CSS.

```
/* docsample/tokensUsage.css*/
.THIS {
    background-color: grey;
}
.THIS.white {
    background-color: white;
}
.THIS .first {
    background-color: token(myBackgroundColor);
}
.THIS .notFirst {
    background-color: blue;
}
```

The `.THIS .first` selector uses `token(myBackgroundColor)` to set the background color for a list item in the markup. `myBackgroundColor` is set to `#ff0000` in `docsampleNamespace.tokens`.

```
<li class="first">I'm first.</li>
```

The second list item in the markup uses the `other` token.

```
<li class="{!token('other')}">I'm second.</li>
```

The value of the `other` token is set in `docsampleNamespace.tokens`.

```
<aura:token name="other" value="notFirst"/>
```

The `.THIS .notFirst` CSS selector sets the background color for this list item to blue.

Finally, let's look at the app that loads the tokens file.

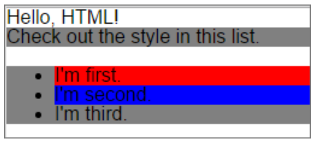
```
<!--docsample:wrapperApp-->
<aura:application tokens="docsample:docsampleNamespace">
    <docsample:tokensUsage />
</aura:application>
```

The `tokens` attribute loads the tokens file at `docsample/docsampleNamespace/docsampleNamespace.tokens`.

After starting your server, navigate to the app at:

`http://localhost:<port>/docsample/wrapperApp.app`

The output should look like this.


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

- I'm first
- I'm second
- I'm third.

SEE ALSO:

Using Expressions

Defining Tokens

Using JavaScript

Use JavaScript for client-side code. The `$A` namespace is the entry point for using the framework in JavaScript code.

For all the methods available in `$A`, see the JavaScript API.

A component bundle can contain JavaScript code in a client-side controller, 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 methods for each JavaScript object. When you are writing code, it's important to understand which methods are publicly accessible for a JavaScript object.

A publicly accessible method is annotated with `@export`. Any method that doesn't have an `@export` annotation is for internal use by the framework.

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.

IN THIS SECTION:

Using External JavaScript Libraries
To use an external JavaScript library in your apps, include a `<script>` tag in your `.app` file or include it in your app's template.

Creating Reusable JavaScript Libraries
An Aura JavaScript library enables you to create a set of JavaScript files that can be used by any component that imports the library.

Working with Attribute Values in JavaScript
These are useful and common patterns for working with attribute values in JavaScript.

Working with a Component Body in JavaScript
These are useful and common patterns for working with a component's body 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.

Modifying the DOM

The Document Object Model (DOM) is the language-independent model for representing and interacting with objects in HTML and XML documents. It's important to know how to modify the DOM safely so that the framework's rendering service doesn't stomp on your changes and give you unexpected results.

Client-Side Rendering to the DOM

The framework's rendering service takes in-memory component state and creates and manages the DOM elements owned by the component. If you want to modify DOM elements created by the framework for a component, modify the DOM elements in the component's renderer. Otherwise, the framework will override your changes when the component is rerendered.

Client-Side Runtime Binding of Components

A provider enables you to use an abstract component in markup. The framework uses the provider to determine the concrete component to use at runtime.

Invoking Actions on Component Initialization

Use the `init` event to initialize a component or fire an event after component construction but before rendering.

Modifying Components Outside the Framework Lifecycle

Use `$A.getCallback()` to wrap any code that modifies a component outside the normal rerendering lifecycle, such as in a `setTimeout()` call. The `$A.getCallback()` call ensures that the framework rerenders the modified component and processes any enqueued actions.

Validating Fields

Validate user input, handle errors, and display error messages on input fields.

Throwing and Handling Errors

The framework gives you flexibility in handling unrecoverable and recoverable app errors in JavaScript code. For example, you can throw these errors in a callback when handling an error in a server-side response.

Calling Component Methods

Use `<aura:method>` to define a method as part of a component's API. This enables you to directly call a method in a component's client-side controller instead of firing and handling a component event. Using `<aura:method>` simplifies the code needed for a parent component to call a method on a child component that it contains.

Using JavaScript Promises

You can use ES6 Promises in JavaScript code. Promises can simplify code that handles the success or failure of asynchronous calls, or code that chains together multiple asynchronous calls.

Making API Calls

You can make API calls from client-side code, but it's not a best practice. Make API calls from server-side controllers instead to maximize performance.

SEE ALSO:

Handling Events with Client-Side Controllers

Using External JavaScript Libraries

To use an external JavaScript library in your apps, include a `<script>` tag in your `.app` file or include it in your app's template.

SEE ALSO:

Creating App Templates

aura:application

Creating Reusable JavaScript Libraries

An Aura JavaScript library enables you to create a set of JavaScript files that can be used by any component that imports the library.

Creating a Library

Every library lives in a namespace and follows a naming convention. The `c:myLib` library points to a library in `c/myLib/myLib.lib`.

A library is defined in a `.lib` file that starts with the `<aura:library>` tag. A library includes an arbitrary number of JavaScript files. Each file is defined with an `<aura:include>` tag.

For example, this `myLib.lib` library includes three files.

```
<aura:library description="A set of global services">
    <aura:include name="ViewService" />
    <aura:include name="ErrorService" />
    <aura:include name="LogService" imports="ErrorService"/>
</aura:library>
```

The `name` attribute of `<aura:include>` is the name of the JavaScript file with or without the `.js` suffix. For example, `name="ViewService"` or `name="ViewService.js"` points to `c/myLib/ViewService.js`.

Note: The `name` attribute can only contain letters, numbers, dot, dash, and underscore characters and must start with a letter or underscore.

The `imports` attribute of `<aura:include>` specifies another JavaScript file that is referenced and imported in the included file. For example, this import specifies that `LogService.js` uses code in `ErrorService.js`.

```
<aura:include name="LogService" imports="ErrorService"/>
```

This example imports an `OtherErrorService` file that is used in a different library, `othernamespace:otherLibrary`.

```
<aura:include name="LogService"
        imports="othernamespace:otherLibrary:OtherErrorService"/>
```

Use a comma-separated list to import more than one JavaScript file. This example imports multiple files.

```
<aura:include name="LogService" imports="ErrorService,BaseLogService"/>
```

Importing a Library

Use the `<aura:import>` tag in a component's markup to import a library and enable usage of the library in the component's JavaScript files.

This markup imports the `c:myLib` library.

```
<aura:import library="c:myLib" property="BaseServices" />
```

The `property` attribute of `<aura:import>` is the variable name that can be used in the component's helper to reference the library.

Using a Library

You can reference the library in the helper of the component that imports the library by using the `property` attribute.

To reference one of the JavaScript files from the library in the helper, use this syntax:

```
var includedFile = this.<property>.<name>;
```

where `<property>` is defined by the `<aura:import>` tag, and `<name>` is defined by the `<aura:include>` tag. For example:

```
var ViewService = this.BaseServices.ViewService;
```

`BaseServices` is the `property` value defined in the `<aura:import>` tag.

To reference a library file in the controller, use this syntax:

```
var ViewService = helper.BaseServices.ViewService;
```

Format of Library Files

This type of library is for libraries stored locally on your filesystem. For external JavaScript libraries, see Using External JavaScript Libraries instead.

Each file in a library must be a function and use a module pattern. The return value of the function is the singleton instance that is bound to the helper of a component that imports the library.

For example, let's look at a sample `ViewService.js` file that imports the `ErrorService.js` and `BaseLogService.js` files.

```
<aura:include name="LogService" imports="ErrorService,BaseLogService" />
```

Here is `ViewService.js`:

```
function(ErrorService, BaseLogService) {
    return {
        getView: function() {
            ...
        }
    }
}
```

Note how the imported `ErrorService.js` and `BaseLogService.js` files are set as parameters in the opening function.

if you want to name the function parameters differently, you can configure aliases. For example:

```
<aura:include name="LogService" imports="ErrorService,BaseLogService"
    aliases="es,bls" />
```

Here is `ViewService.js` using the aliases:

```
function(es, bls) {
    return {
        getView: function() {
            ...
        }
    }
}
```

The `aliases` are variables available in the lexical scope of the module and correspond one-to-one to the list of `imports`.

You can call the `getView()` function in a helper using this syntax:

```
var view = this.BaseServices.ViewService.getView();
```

Shim for Library Files

What if you have a library file that doesn't match the required format where each file must be a function? The framework can create a shim to wrap a file in the required format. The shim enables you to use third-party code without having to modify it, which makes maintenance and upgrades easier.

For example, if you have a local copy of the Backbone library, you can create a file to manually wrap the code.

```
function($, _) {
    // Original backbone code
    return Backbone;
}
```

A better approach is to let the framework create the shim to wrap it in the required format.

```
<aura:include name="Backbone" imports="jQuery, Underscore"
    aliases="$, _" export="Backbone"/>
```

The `export` attribute corresponds to the variable that exists in the lexical scope that you use in your helper.

Working with Attribute Values in JavaScript

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

`component.get(String key)` and `component.set(String key, Object value)` retrieves and assigns values associated with the specified key on the component. Keys are passed in as an expression, which represents attribute values. To retrieve an attribute value of a component reference, use `component.find("cmpId").get("v.value")`. Similarly, use `component.find("cmpId").set("v.value", myValue)` to set the attribute value of a component reference. This example shows how you can retrieve and set attribute values on a component reference, represented by the button with an ID of `button1`.

```
<aura:component>
    <aura:attribute name="buttonLabel" type="String"/>
    <ui:button aura:id="button1" label="Button 1"/>
    {!v.buttonLabel}
    <ui:button label="Get Label" press="{!c.getLabel}"/>
</aura:component>
```

This controller action retrieves the `label` attribute value of a button in a component and sets its value on the `buttonLabel` attribute.

```
({
    getLabel : function(component, event, helper) {
        var myLabel = component.find("button1").get("v.label");
        component.set("v.buttonLabel", myLabel);
    }
})
```

In the following examples, `cmp` is a reference to a component in your JavaScript code.

Get an Attribute Value

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

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

Set an Attribute Value

To set the value of a component's `label` attribute:

```
cmp.set("v.label","This is a label");
```

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. Remember that the `body` attribute is an array of components, so you can use the JavaScript `Array` methods on it.

Note: When you use `cmp.set("v.body", ...)` to set the component body, you must explicitly include `{!v.body}` in your component markup.

Replace a Component's Body

To replace the current value of a component's body with another component:

```
// newCmp is a reference to another component
cmp.set("v.body", newCmp);
```

Clear a Component's Body

To clear or empty the current value of a component's body:

```
cmp.set("v.body", []);
```

Append a Component to a Component's Body

To append a `newCmp` component to a component's body:

```
var body = cmp.get("v.body");
// newCmp is a reference to another component
body.push(newCmp);
cmp.set("v.body", body);
```

Prepend a Component to a Component's Body

To prepend a `newCmp` component to a component's body:

```
var body = cmp.get("v.body");
body.unshift(newCmp);
cmp.set("v.body", body);
```

Remove a Component from a Component's Body

To remove an indexed entry from a component's body:

```
var body = cmp.get("v.body");
// Index (3) is zero-based so remove the fourth component in the body
body.splice(3, 1);
cmp.set("v.body", body);
```

SEE ALSO:

Component Body

Working with Attribute Values in JavaScript

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 any JavaScript code in a component's bundle, such as from a client-side controller or renderer. Helper functions are similar to client-side controller functions in shape, surrounded by brackets and curly braces to denote a JSON object containing a map of name-value pairs. A helper function can pass in any arguments required by the function, such as the component it belongs to, a callback, or any other objects.

Creating a Helper

A helper file is part of the component bundle and is auto-wired via 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. You can also pass in an instance variable as a parameter, for example, `createExpense: function(component, expense){...}`, where `expense` is a variable defined in the component.

The following code shows you how to call the `updateItem` helper function in a controller, which can be used with a custom event handler.

```
({
    newItemEvent: function(component, event, helper) {
        helper.updateItem(component, event.getParam("item"));
    }
})
```

Helper functions are local to a component, improve code reuse, and move the heavy lifting of JavaScript logic away from the client-side controller where possible. The following code shows the helper function, which takes in the `value` parameter set in the controller via the `item` argument. The code walks through calling a server-side action and returning a callback but you can do something else in the helper function.

```
({
    updateItem : function(component, item, callback) {
        //Update the items via a server-side action
        var action = component.get("c.saveItem");
        action.setParams({"item" : item});
        //Set any optional callback and enqueue the action
        if (callback) {
            action.setCallback(this, callback);
        }
        $A.enqueueAction(action);
    }
})
```

SEE ALSO:

- Client-Side Rendering to the DOM
- Component Bundles
- Handling Events with Client-Side Controllers

Modifying the DOM

The Document Object Model (DOM) is the language-independent model for representing and interacting with objects in HTML and XML documents. It's important to know how to modify the DOM safely so that the framework's rendering service doesn't stomp on your changes and give you unexpected results.

The framework's rendering service takes in-memory component state and creates and manages the DOM elements owned by the component. The framework automatically renders your components so you don't have to know anything more about rendering unless you need to customize the default rendering behavior for a component.

There are a few supported ways to modify the DOM.

DOM Elements Managed by Aura

The framework creates and manages the DOM elements owned by a component. If you want to modify these DOM elements created by the framework, modify the DOM elements in the component's renderer. Otherwise, the framework will override your changes when the component is rerendered.

For example, if you modify DOM elements directly from a client-side controller, the changes may be overwritten when the components are rendered. Instead, update the component's attributes and let the framework's rendering service take care of the DOM updates.

You don't normally have to write a custom renderer, but it's useful if you need to interact with the DOM tree after the framework's rendering service has inserted DOM elements. If you want to customize rendering behavior and you can't do it in markup or by using the `init` event, create a client-side renderer.

In a renderer, modify the DOM that belongs to the current component only. Never break component encapsulation by reaching into another component and changing its DOM elements, even if you are reaching in from the parent component.

There are often better alternatives to creating a custom renderer. Consider using an expression in the markup instead of setting a DOM element directly.

You can modify CSS classes for a component outside a renderer by using the `$A.util.addClass()`, `$A.util.removeClass()`, and `$A.util.toggleClass()` methods.

You can read from the DOM outside a renderer.

DOM Elements Managed by External Libraries

You can use different libraries, such as a charting library, to create and manage DOM elements. You don't have to modify these DOM elements within a renderer. A renderer is only used to customize DOM elements created and managed by Aura.

SEE ALSO:

Client-Side Rendering to the DOM

Using Expressions

Invoking Actions on Component Initialization

Dynamically Showing or Hiding Markup

Client-Side Rendering to the DOM

The framework's rendering service takes in-memory component state and creates and manages the DOM elements owned by the component. If you want to modify DOM elements created by the framework for a component, modify the DOM elements in the component's renderer. Otherwise, the framework will override your changes when the component is rerendered.

The DOM is the language-independent model for representing and interacting with objects in HTML and XML documents. The framework automatically renders your components so you don't have to know anything more about rendering unless you need to customize the default rendering behavior for a component.

For more details on whether creating a custom renderer is the right choice, see Modifying the DOM.

Base Component Rendering

The base component in the framework is `aura:component`. Every component extends this base component.

The renderer for `aura:component` is in `componentRenderer.js`. This renderer has base implementations for the four phases of the rendering and rerendering cycles:

- `render()`
- `rerender()`
- `afterRender()`
- `unrender()`

The framework calls these functions as part of the rendering and rerendering lifecycles and we will learn more about them soon. You can override the base rendering functions in a custom renderer.

Rendering Lifecycle

The rendering lifecycle happens once in the lifetime of a component unless the component gets explicitly unrendered. When you create a component:

1. The framework fires an `init` event, enabling you to update a component or fire an event after component construction but before rendering.
2. The `render()` method is called to render the component's body.
3. The `afterRender()` method is called to enable you to interact with the DOM tree after the framework's rendering service has inserted DOM elements.

Rerendering Lifecycle

The rerendering lifecycle automatically handles rerendering of components whenever the underlying data changes. Here is a typical sequence.

1. A browser event triggers one or more 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 by calling each component's `rerender()` method.

The component rerendering lifecycle repeats whenever the underlying data changes as long as the component is valid and not explicitly unrendered.

For more information, see Events Fired During the Rendering Lifecycle .

Create a Renderer

You don't normally have to write a custom renderer, but it's useful when you want to interact with the DOM tree after the framework's rendering service has inserted DOM elements. If you want to customize rendering behavior and you can't do it in markup or by using the `init` event, you can create a client-side renderer.

A renderer file is part of the component bundle and is auto-wired if you follow the naming convention, `<componentName>Renderer.js`. For example, the renderer for `sample.cmp` would be in `sampleRenderer.js`.

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 `docsample.sampleComponent` in `docsample/sampleComponent/sampleComponentRenderer.js`.

```
<aura:component
    renderer="js://docsample.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.

Note: These guidelines are important when you customize rendering.

- Only modify DOM elements that are part of the component. Never break component encapsulation by reaching in to another component and changing its DOM elements, even if you are reaching in from the parent component.

- Never fire an event as it can trigger new rendering cycles. An alternative is to use an `init` event instead.
- Don't set attribute values on other components as these changes can trigger new rendering cycles.
- Move as much of the UI concerns, including positioning, to CSS.

Customize Component Rendering

Customize rendering by creating a `render()` function in your component's renderer to override the base `render()` function, which updates the DOM.

The `render()` function returns a DOM node, an array of DOM nodes, or nothing. The base HTML component expects DOM nodes when it renders a component.

You generally want to extend default rendering by calling `superRender()` from your `render()` function before you add your custom rendering code. Calling `superRender()` creates the DOM nodes specified in the markup.

This code outlines a custom `render()` function.

```
render : function(cmp, helper) {
    var ret = this.superRender();
    // do custom rendering here
    return ret;
},
```

Rerender Components

When an event is fired, it may trigger actions to change data and call `rerender()` on affected components. The `rerender()` function enables components to update themselves based on updates to other components since they were last rendered. This function doesn't return a value.

If you update data in a component, the framework automatically calls `rerender()`.

You generally want to extend default rerendering by calling `superRerender()` from your `renderer()` function before you add your custom rerendering code. Calling `superRerender()` chains the rerendering to the components in the `body` attribute.

This code outlines a custom `rerender()` function.

```
rerender : function(cmp, helper){
    this.superRerender();
    // do custom rerendering here
}
```

Access the DOM After Rendering

The `afterRender()` function enables you to interact with the DOM tree after the framework's rendering service has inserted DOM elements. It's not necessarily the final call in the rendering lifecycle; it's simply called after `render()` and it doesn't return a value.

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

This code outlines a custom `afterRender()` function.

```
afterRender: function (component, helper) {
    this.superAfterRender();
    // interact with the DOM here
},
```

Unrender Components

The base `unrender()` function deletes all the DOM nodes rendered by a component's `render()` function. It is called by the framework when a component is being destroyed. Customize this behavior by overriding `unrender()` in your component's renderer. This method can be useful when you are working with third-party libraries that are not native to the framework.

You generally want to extend default unrendering by calling `superUnrender()` from your `unrender()` function before you add your custom code.

This code outlines a custom `unrender()` function.

```
unrender: function () {
    this.superUnrender();
    // do custom unrendering here
}
```

Ensure Client-Side Rendering

The framework calls the default server-side renderer by default, or a client-side renderer if you have one. To ensure client-side rendering of a top-level component, append `render="client"` to the `aura:component` tag. Setting `render="client"` in the top-level component takes precedence over the framework's detection logic, which takes dependencies into consideration. This behavior is useful if you're testing the component 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.

SEE ALSO:

- Modifying the DOM
- Invoking Actions on Component Initialization
- Component Bundles
- Modifying Components Outside the Framework Lifecycle
- Sharing JavaScript Code in a Component Bundle
- Server-Side Rendering to the DOM

Client-Side Runtime Binding of Components

A provider enables you to use an abstract component 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 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.

SEE ALSO:

Server-Side Runtime Binding of Components

Abstract Components

Interfaces

Component Bundles

aura:dependency

Invoking Actions on Component Initialization

Use the `init` event to initialize 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}"/>

    <p>This value is set in the controller after the component initializes and before
rendering.</p>
    <p><b>{!v.setMeOnInit}</b></p>

</aura:component>
```

Client-side controller source

```
({
    doInit: function(cmp) {
        // Set the attribute value.
        // You could also 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 handler for the component. `init` is a predefined event sent to every 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 with Change Handlers

Modifying Components Outside the Framework Lifecycle

Use `$A.getCallback()` to wrap any code that modifies a component outside the normal rerendering lifecycle, such as in a `setTimeout()` call. The `$A.getCallback()` call ensures that the framework rerenders the modified component and processes any enqueued actions.

Note: `$A.run()` is deprecated. Use `$A.getCallback()` instead.

You don't need to use `$A.getCallback()` if your code is executed as part of the framework's call stack; for example, your code is handling an event or in the callback for a server-side controller action.

An example of where you need to use `$A.getCallback()` is calling `window.setTimeout()` in an event handler to execute some logic after a time delay. This puts your code outside the framework's call stack.

This sample sets the `visible` attribute on a component to `true` after a five-second delay.

```
window.setTimeout(
    $A.getCallback(function() {
        if (cmp.isValid()) {
            cmp.set("v.visible", true);
        }
    }), 5000
);
```

Note how the code updating a component attribute is wrapped in `$A.getCallback()`, which ensures that the framework rerenders the modified component.

Note: Always add an `isValid()` check if you reference a component in asynchronous code, such as a callback or a timeout. If you navigate elsewhere in the UI while asynchronous code is executing, the framework unrenders and destroys the component that made the asynchronous request. You can still have a reference to that component, but it is no longer valid. Add an `isValid()` call to check that the component is still valid before processing the results of the asynchronous request.

Warning: Don't save a reference to a function wrapped in `$A.getCallback()`. If you use the reference later to send actions, the saved transaction state will cause the actions to be aborted.

SEE ALSO:

Handling Events with Client-Side Controllers

Firing Aura Events from Non-Aura Code

Communicating with Events

Validating Fields

Validate user input, handle errors, and display error messages on input fields.

Client-side input validation is available for the following components:

- `lightning:input`
- `lightning:select`
- `lightning:textarea`
- `ui:input*`

Components in the `lightning` namespace simplify input validation by providing attributes to define error conditions, enabling you to handle errors by checking the component's validity state. For example, you can set a minimum length for a field , display an error message when the condition is not met, and handle the error based on the given validity state. Alternatively, input components in the `ui` namespace let you define and handle errors in a client-side controller. See the `lightning` namespace components in the Reference section for more information.

The following sections discuss error handling for `ui:input*` components.

Default Error Handling

The framework can handle and display errors using the default error component, `ui:inputDefaultError`. The following example shows how the framework handles a validation error and uses the default error component to display the error message. Here is the markup.

```
<!--c:errorHandling-->
<aura:component>
    Enter a number: <ui:inputNumber aura:id="inputCmp"/> <br/>
    <ui:button label="Submit" press="{!c.doAction}"/>
</aura:component>
```

Here is the client-side controller.

```
/*errorHandlingController.js*/
{
    doAction : function(component) {
        var inputCmp = component.find("inputCmp");
        var value = inputCmp.get("v.value");

        // Is input numeric?
        if (isNaN(value)) {
            // Set error
            inputCmp.set("v.errors", [{message:"Input not a number: " + value}]);
        } else {
            // Clear error
            inputCmp.set("v.errors", null);
        }
    }
}
```

When you enter a value and click **Submit**, `doAction` 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. Add error messages to the input component using the `errors` attribute.

Custom Error Handling

`ui:input` and its child components can handle errors using the `onError` and `onClearErrors` events, 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 following example shows how you can handle a validation error using custom error handlers and display the error message using the default error component. Here is the markup.

```
<!--c:errorHandlingCustom-->
<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>
```

Here is the client-side controller.

```
/*errorHandlingCustomController.js*/
{
    doAction : function(component, event) {
        var inputCmp = component.find("inputCmp");
        var value = inputCmp.get("v.value");

        // is input numeric?
        if (isNaN(value)) {
            inputCmp.set("v.errors", [{message:"Input not a number: " + value}]);
        } else {
            inputCmp.set("v.errors", null);
        }
    },

    handleError: function(component, event){
        /* do any custom error handling
         * logic desired here */
        // get v.errors, which is an Object[]
        var errorsArr  = event.getParam("errors");
        for (var i = 0; i < errorsArr.length; i++) {
            console.log("error " + i + ": " + JSON.stringify(errorsArr[i]));
        }
    },

    handleClearError: function(component, event) {
        /* do any custom error handling
         * logic desired here */
    }
}
```

When you enter a value and click **Submit**, `doAction` in the controller executes. However, instead of letting the framework handle the errors, we define a custom error handler using the `onError` event in `<ui:inputNumber>`. If the validation fails, `doAction` adds an error message using the `errors attribute`. This automatically fires the `handleError` custom error handler.

Similarly, you can customize clearing 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 and Handling Errors

The framework gives you flexibility in handling unrecoverable and recoverable app errors in JavaScript code. For example, you can throw these errors in a callback when handling an error in a server-side response.

Unrecoverable Errors

Use `throw new Error("error message here")` for unrecoverable errors, such as an error that prevents your app from starting successfully. The error message and a stack trace are displayed.

Note: `$A.error()` is deprecated. Throw the native JavaScript `Error` object instead by using `throw new Error()`.

This example shows you the basics of throwing an unrecoverable error in a JavaScript controller.

```
<!--c:unrecoverableError-->
<aura:component>
    <ui:button label="throw error" press="{!c.throwError}"/>
</aura:component>
```

Here is the client-side controller source.

```
/*unrecoverableErrorController.js*/
({
    throwError : function(component, event){
        throw new Error("I can’t go on. This is the end.");
    }
})
```

Recoverable Errors

Throw an instance of `$A.auraFriendlyError()` for recoverable errors.

This example shows you the basics of throwing and handling an error in a JavaScript controller.

```
<!--c:recoverableError-->
<aura:component>
    <aura:handler event="aura:systemError" action="{!c.showError}"/>

    <ui:button label="throw error" press="{!c.throwError}"/>
    <div aura:id="myError" class="isDisplayed message" >
        <ui:outputText aura:id="message" value=""/>
        <ui:button label="OK" press="{!c.hideError}"/>
    </div>
</aura:component>
```

Click **throw error** to call `throwError` in the client-side controller.

In this simple example, we display an error message using a `ui:outputText` component. Typically, you use a component, such as `ui:message` or `ui:panel`, to tell the user about the problem. Another button labeled **OK** lets you dismiss the error.

Here is the client-side controller source.

```
/*recoverableErrorController.js*/
({
    throwError : function(cmp, event){
        // error is an instance of AuraFriendlyError
        // argument sets the message property of AuraFriendlyError
        var error = new $A.auraFriendlyError("This is a sample error.");
        // set an optional error data object
        error.data = {
            "moreErrorData1": "more1",
            "moreErrorData2": "more2",
        };
        throw error;
    },

    showError: function(cmp, event) {
        // handle the error by displaying a message
        var myErrorCmp = cmp.find('myError');
        var messageCmp = cmp.find('message');

        // get the error object from aura:systemError event
        // This is the AuraFriendlyError object
        var afe = event.getParam('auraError');
        if (afe) {
            // message is the argument set in AuraFriendlyError
            var message = afe.message;
            $A.util.removeClass(myErrorCmp, "isDisplayed");
            messageCmp.set("v.value", message
                + JSON.stringify(afe.data));
        }
        // aura:systemError event has been handled
        event["handled"] = true;
    },

     hideError : function(cmp){
        // hide the error message from view
        var myErrorCmp = cmp.find('myError');
        $A.util.addClass(myErrorCmp,"isDisplayed");
    }

})
```

The `throwError` method throws an instance of `$A.auraFriendlyError()`, which triggers firing of the `aura:systemError` event.

Set the error message in the `message` property of `AuraFriendlyError`, which corresponds to the first argument of `$A.auraFriendlyError()`. Set an optional object with more context in the `data` property of `AuraFriendlyError`.

An `aura:systemError` event handler in the markup calls `showError` to handle the error.

Set `event["handled"]=true` in `showError` to indicate that you're providing your own error handler for the `aura:systemError` event.

Here is the CSS.

```
/*recoverableError.css*/
.THIS.isDisplayed {
    display: none;
}
```

SEE ALSO:

Validating Fields

Calling Component Methods

Use `<aura:method>` to define a method as part of a component's API. This enables you to directly call a method in a component's client-side controller instead of firing and handling a component event. Using `<aura:method>` simplifies the code needed for a parent component to call a method on a child component that it contains.

Use this syntax to call a method in JavaScript code.

```
cmp.sampleMethod(arg1, … argN);
```

`cmp` is a reference to the component. `arg1, … argN` is an optional comma-separated list of arguments passed to the method.

Let's look at an example of a component containing a button. The handler for the button calls a component method instead of firing and handling its own component event.

Here is the component source.

```
<!--c:auraMethod-->
<aura:component>
    <aura:method name="sampleMethod" action="{!c.doAction}"
      description="Sample method with parameters">
        <aura:attribute name="param1" type="String" default="parameter 1" />
    </aura:method>

    <ui:button label="Press Me" press="{!c.handleClick}"/>
</aura:component>
```

Here is the client-side controller.

```
/*auraMethodController.js*/
({
    handleClick : function(cmp, event) {
        console.log("in handleClick");
        // call the method declared by <aura:method> in the markup
        cmp.sampleMethod("1");
    },

    doAction : function(cmp, event) {
        var params = event.getParam('arguments');
        if (params) {
            var param1 = params.param1;
            console.log("param1: " + param1);
            // add your code here
        }
    },
})
```

This simple example just logs the parameter passed to the method.

The `<aura:method>` tag set `name="sampleMethod"` and `action="{!c.doAction}"` so the method is called by `cmp.sampleMethod()` and handled by `doAction()` in the controller.

Note: If you don't specify an `action` value, the controller action defaults to the value of the method `name`. If we omitted `action="{!c.doAction}"` from the earlier example, the method would be called by `cmp.sampleMethod()` and handled by `sampleMethod()` instead of `doAction()` in the controller.

Using Inherited Methods

A sub component that extends a super component has access to any methods defined in the super component.

An interface can also include an `<aura:method>` tag. A component that implements the interface can access the method.

SEE ALSO:

aura:method

Component Events

Using JavaScript Promises

You can use ES6 Promises in JavaScript code. Promises can simplify code that handles the success or failure of asynchronous calls, or code that chains together multiple asynchronous calls.

If the browser doesn't provide a native version, the framework uses a polyfill so that promises work in all browsers supported for Lightning Experience.

We assume that you are familiar with the fundamentals of promises. For a great introduction to promises, see https://developers.google.com/web/fundamentals/getting-started/primers/promises.

Promises are an optional feature. Some people love them, some don't. Use them if they make sense for your use case.

When you need to coordinate or chain together multiple callbacks, promises can be useful . The generic pattern is:

```
somePromise()
    .then(
        // resolve handler
        $A.getCallback(function(result) {
            return anotherPromise();
        }),

        // reject handler
        $A.getCallback(function(error) {
            console.log("Promise was rejected: ", error);
            return errorRecoveryPromise();
        })
    )
    .then(
        // resolve handler
        $A.getCallback(function() {
            return yetAnotherPromise();
        })
    );
```

`somePromise()` returns a `Promise`. The constructor in that promise determines the conditions for calling `resolve()` or `reject()` on the promise.

The `then()` method chains multiple promises. In this example, each resolve handler returns another promise.

`then()` is part of the Promises API. It takes two arguments:

1. A callback for a fulfilled promise (resolve handler)
2. A callback for a rejected promise (reject handler)

The first callback, `function(result)`, is called when `resolve()` is called in the promise constructor. The `result` object in the callback is the object passed as the argument to `resolve()`.

The second callback, `function(error)`, is called when `reject()` is called in the promise constructor. The `error` object in the callback is the object passed as the argument to `reject()`.

Note: The two callbacks are wrapped by `$A.getCallback()` in our example. What's that all about? Promises execute their resolve and reject functions asynchronously so the code is outside the Aura event loop and normal rendering lifecycle. If the resolve or reject code makes any calls to Aura, such as setting a component attribute, use `$A.getCallback()` to wrap the code. For more information, see Modifying Components Outside the Framework Lifecycle on page 155.

Always Use `catch()` or a Reject Handler

The reject handler in the first `then()` method returns a promise with `errorRecoveryPromise()`. Reject handlers are often used "midstream" in a promise chain to trigger an error recovery mechanism.

The Promises API includes a `catch()` method to optionally catch unhandled errors. Always include a reject handler or a `catch()` method in your promise chain.

Throwing an error in a promise doesn't trigger `window.onerror`, which is where the framework configures its global error handler. If you don't have a `catch()` method, keep an eye on your browser's console during development for reports about uncaught errors in a promise. To show an error message in a `catch()` method, use `$A.reportError()`. The syntax for `catch()` is:

```
promise.then(...)
    .catch(function(error) {
        $A.reportError("error message here", error);
    });
```

For more information on `catch()`, see the Mozilla Developer Network.

Don't Use Storable Actions in Promises

The framework stores the response for storable actions in client-side cache. This stored response can dramatically improve the performance of your app and allow offline usage for devices that temporarily don't have a network connection. Storable actions are only suitable for read-only actions.

Storable actions might have their callbacks invoked more than once: first with cached data, then with updated data from the server. This doesn't align well with promises, which are expected to resolve or reject only once.

Using Promises in Tests

You can return a promise from the current test stage and the test framework will wait for that promise to resolve or reject before continuing to the next test stage or completing the test. You don't need any boilerplate code to handle errors or wait for promises to complete.

Here's an example:

```
testThatUsesPromises: {
    test: function(cmp) {
        return somePromise()
            .then(function() {
                return anotherPromise();
            });
    }
}
```

SEE ALSO:

Storable Actions

Making API Calls

You can make API calls from client-side code, but it's not a best practice. Make API calls from server-side controllers instead to maximize performance.

The framework uses an `XMLHttpRequest` (XHR) to communicate from the client to the server and server-side actions are designed to minimize network traffic and provide a smoother user experience.

Batching of Actions

The framework queues up actions before sending them to the server. This mechanism is largely transparent to you when you're writing code but it enables the framework to minimize network traffic by batching multiple actions into one request. For more information, see Queueing of Server-Side Actions on page 183.

Abortable Actions

Mark an action as abortable to make it potentially abortable while it's queued to be sent to the server. An abortable action in the queue is not sent to the server if the component that created the action is no longer valid, that is `cmp.isValid() == false`. A component is automatically destroyed and marked invalid by the framework when it is unrendered. For more information, see Abortable Actions on page 184.

Storable Actions

Mark an action as storable to have its response stored in the client-side cache by the framework. Caching can be useful if you want your app to be functional for devices that temporarily don't have a network connection. For more information, see Storable Actions on page 186.

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. A rough guideline is to use a background action if it takes more than five seconds for the response to return from the server. For more information, see Foreground and Background Actions on page 183.

JavaScript Cookbook

This section includes code snippets and samples that can be used in various JavaScript files.

IN THIS SECTION:

Adding and Removing Styles
You can add or remove a CSS style on a component or element during runtime.

Which Button Was Pressed?
To find out which button was pressed in a component containing multiple buttons, use `Component.getLocalId()`.

Dynamically Creating Components

Create a component dynamically in your client-side JavaScript code by using the `$A.createComponent()` method. To create multiple components, use `$A.createComponents()`.

Note: Use `$A.createComponent()` instead of the deprecated `newComponent()`, `newComponentAsync()`, and `newComponentDeprecated()` methods of `AuraComponentService` or the deprecated `newCmp()` and `newCmpAsync()` methods of `$A`.

The syntax is:

```
$A.createComponent(String type, Object attributes, function callback)
```

1. `type`—The type of component to create; for example, `"ui:button"`.
2. `attributes`—A map of attributes for the component, including the local Id (`aura:id`) and flavor (`aura:flavor`).
3. `callback(cmp, status, errorMessage)`—The callback to invoke after the component is created. The callback has three parameters.
   a. `cmp`—The new component created. This enables you to do something with the new component, such as add it to the body of the component that creates it. If there's an error, `cmp` is `null`.
   b. `status`—The status of the call. The possible values are `SUCCESS`, `INCOMPLETE`, or `ERROR`. Always check the status is `SUCCESS` before you try to use the component.
   c. `errorMessage`—The error message if the status is `ERROR`.

Let's add a dynamically created button to this sample component.

```
<!--c:createComponent-->
<aura:component>
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <p>Dynamically created button</p>
    {!v.body}
</aura:component>
```

The client-side controller calls `$A.createComponent()` to create a `ui:button` with a local ID and a handler for the `press` event. The `function(newButton, ...)` callback appends the button to the `body` of `c:createComponent`. The `newButton` that's dynamically created by `$A.createComponent()` is passed as the first argument to the callback.

```
/*createComponentController.js*/
({
    doInit : function(cmp) {
        $A.createComponent(
            "ui:button",
            {
                "aura:id": "findableAuraId",
                "aura:flavor": "default, neutral",
                "label": "Press Me",
                "press": cmp.getReference("c.handlePress")
            },
            function(newButton, status, errorMessage){
                //Add the new button to the body array
                if (status === "SUCCESS") {
                    var body = cmp.get("v.body");
                    body.push(newButton);
                    cmp.set("v.body", body);
                }
                else if (status === "INCOMPLETE") {
                    console.log("No response from server or client is offline.")
                    // Show offline error
                }
                else if (status === "ERROR") {
                    console.log("Error: " + errorMessage);
                    // Show error message
                }
            }
        );
    },

    handlePress : function(cmp) {
        console.log("button pressed");
    }
})
```

Note: `c:createComponent` contains a `{!v.body}` expression. When you use `cmp.set("v.body", ...)` to set the component body, you must explicitly include `{!v.body}` in your component markup.

Creating Nested Components

To dynamically create a component in the body of another component, use `$A.createComponents()` to create the components. In the function callback, nest the components by setting the inner component in the `body` of the outer component. This example creates a `ui:outputText` component in the `body` of a `ui:message` component.

```
$A.createComponents([
    ["ui:message",{
        "title" : "Sample Thrown Error",
        "severity" : "error",
    }],
    ["ui:outputText",{
        "value" : e.message
    }]
    ],
    function(components, status, errorMessage){
        if (status === "SUCCESS") {
            var message = components[0];
            var outputText = components[1];
            // set the body of the ui:message to be the ui:outputText
            message.set("v.body", outputText);
        }
        else if (status === "INCOMPLETE") {
            console.log("No response from server or client is offline.")
            // Show offline error
        }
        else if (status === "ERROR") {
            console.log("Error: " + errorMessage);
            // Show error message
        }
    }
);
```

Destroying Dynamically Created Components

After a component that is declared in markup is no longer in use, the framework automatically destroys it and frees up its memory.

If you create a component dynamically in JavaScript and that component isn't added to a facet (`v.body` or another attribute of type `Aura.Component[]`), you have to destroy it manually using `Component.destroy()` to avoid memory leaks.

Avoiding a Server Trip

The `createComponent()` and `createComponents()` methods supports both client-side and server-side component creation. If no server-side dependencies are found, the methods are executed client-side.

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. If there are no server dependencies and the definition already exists on the client via preloading or declared dependencies, no server call is made.

Tip: There's no limit in component creation on the client side. You can create up to 10,000 components in one server request. If you hit this limit, ensure that you're creating components on the client side in markup or in JavaScript using `$A.createComponent()` or `$A.createComponents()`. To avoid a trip to the server for component creation in JavaScript code, add an `<aura:dependency>` tag for the component in the markup to explicitly tell the framework about the dependency.

The framework automatically tracks dependencies between definitions, such as components. However, some dependencies aren't easily discoverable by the framework; for example, if you dynamically create a component that isn't 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.

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.

SEE ALSO:

- Reference Doc App
- aura:dependency
- Invoking Actions on Component Initialization
- Dynamically Adding Event Handlers
- Styling with Flavors

Detecting Data Changes with Change Handlers

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

When the value changes, the `valueChange.evt` event is automatically fired. The event has `type="VALUE"`.

In the component, define a handler with `name="change"`.

```
<aura:handler name="change" value="{!v.numItems}" action="{!c.itemsChange}"/>
```

The `value` attribute sets the component attribute that the change handler tracks.

The `action` attribute sets the client-side controller action to invoke when the attribute value changes.

A component can have multiple `<aura:handler name="change">` tags to detect changes to different attributes.

In the controller, define the action for the handler.

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

The `valueChange` event gives you access to the previous value (`oldValue`) and the current value (`value`) in the handler action.

When a change occurs to a value that is represented by the `change` handler, the framework handles the firing of the event and rerendering of the component.

SEE ALSO:

Invoking Actions on Component Initialization

aura:valueChange

Finding Components by ID

Retrieve a component by its ID in JavaScript code.

Use `aura:id` to add a local ID of `button1` to the `ui:button` component.

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

You can find the component by calling `cmp.find("button1")`, where `cmp` is a reference to the component containing the button. The `find()` function has one parameter, which is the local ID of a component within the markup.

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

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

SEE ALSO:

Component IDs

Value Providers

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 `c:sampleComponent`.

```
addNewHandler : function(cmp, event) {
    var cmpArr = cmp.find({ instancesOf : "c:sampleComponent" });
    for (var i = 0; i < cmpArr.length; i++) {
        var outputCmpArr = cmpArr[i];
        outputCmpArr.addHandler("cmpEvent", cmp, "c.someAction");
    }
}
```

Let's look at the `addHandler()` method that adds an event handler to a component.

```
outputCmpArr.addHandler("cmpEvent", cmp, "c.someAction");
```

- `cmpEvent`—The first argument is the name of the event that triggers the handler. Note that you can't force a component to start firing events that it doesn't fire so make sure that this argument corresponds to an event that the component fires. The `<aura:registerEvent>` tag in a component's markup advertises an event that the component fires. Set this argument to match the `name` attribute of one of the `<aura:registerEvent>` tags.
- `cmp`—The second argument is the value provider for resolving the action expression, which is the next argument. In this example, the value provider is the component associated with the controller.
- `c.someAction`—The third argument is the controller action that handles the event. This is equivalent to the value you would put in the `action` attribute in the `<aura:handler>` tag if the handler was statically defined in the markup.

For a full list of methods and arguments, refer to the JavaScript API in the doc reference app.

You can also add an event handler to a component that is created dynamically in the callback function of `$A.createComponent()`. For more information, see Dynamically Creating Components.

SEE ALSO:

Handling Events with Client-Side Controllers

Handling Component Events

Reference Doc App

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()`.

Dynamically Showing or Hiding Markup

Use CSS to toggle markup visibility. You could use the `<aura:if>` tag to do the same thing but we recommend using CSS as it's the more standard approach.

This example uses `$A.util.toggleClass(cmp, 'class')` to toggle visibility of markup.

```
<!--c:toggleCss-->
<aura:component>
    <ui:button label="Toggle" press="{!c.toggle}"/>
    <p aura:id="text">Now you see me</p>
</aura:component>
```

```
/*toggleCssController.js*/
({
    toggle : function(component, event, helper) {
        var toggleText = component.find("text");
        $A.util.toggleClass(toggleText, "toggle");
    }
})
```

```
/*toggleCss.css*/
.THIS.toggle {
    display: none;
}
```

Click the **Toggle** button to hide or show the text by toggling the CSS class.

SEE ALSO:

Handling Events with Client-Side Controllers

Component Attributes

Adding and Removing Styles

Adding and Removing Styles

You can add or remove a CSS style on a component or element during runtime.

To retrieve the class name on a component, use `component.find('myCmp').get('v.class')`, where `myCmp` is the `aura:id` attribute value.

To append and remove CSS classes from a component or element, use the `$A.util.addClass(cmpTarget, 'class')` and `$A.util.removeClass(cmpTarget, 'class')` methods.

Component source

```
<aura:component>
    <div aura:id="changeIt">Change Me!</div><br />
    <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 cmpTarget = cmp.find('changeIt');
        $A.util.addClass(cmpTarget, 'changeMe');
    },

    removeCSS: function(cmp, event) {
        var cmpTarget = cmp.find('changeIt');
        $A.util.removeClass(cmpTarget, 'changeMe');
    }
}
```

The buttons in this demo are wired to controller actions that append or remove the CSS styles. To append a CSS style to a component, use `$A.util.addClass(cmpTarget, 'class')`. Similarly, remove the class by using `$A.util.removeClass(cmpTarget, 'class')` in your controller. `cmp.find()` locates the component using the local ID, denoted by `aura:id="changeIt"` in this demo.

Toggling a Class

To toggle a class, use `$A.util.toggleClass(cmp, 'class')`, which adds or removes the class.

The `cmp` parameter can be component or a DOM element.

Note: We recommend using a component instead of a DOM element. If the utility function is not used inside `afterRender()` or `rerender()`, passing in `cmp.getElement()` might result in your class not being applied when the components are rerendered. For more information, see Events Fired During the Rendering Lifecycle on page 122.

To hide or show markup dynamically, see Dynamically Showing or Hiding Markup on page 170.

To conditionally set a class for an array of components, pass in the array to `$A.util.toggleClass()`.

```
mapClasses: function(arr, cssClass) {
    for(var cmp in arr) {
        $A.util.toggleClass(arr[cmp], cssClass);
    }
}
```

SEE ALSO:

- Handling Events with Client-Side Controllers
- CSS in Components
- Component Bundles

Which Button Was Pressed?

To find out which button was pressed in a component containing multiple buttons, use `Component.getLocalId()`.

Let's look at a component that contains multiple buttons. Each button has a unique local ID, set by an `aura:id` attribute.

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

    <p>You clicked: {!v.whichButton}</p>

    <ui:button aura:id="button1" label="Click me" press="{!c.nameThatButton}"/>
    <ui:button aura:id="button2" label="Click me too" press="{!c.nameThatButton}"/>
</aura:component>
```

Use `event.getSource()` in the client-side controller to get the button component that was clicked. Call `getLocalId()` to get the `aura:id` of the clicked button.

```
/* buttonPressedController.js */
({
    nameThatButton : function(cmp, event, helper) {
        var whichOne = event.getSource().getLocalId();
        console.log(whichOne);
        cmp.set("v.whichButton", whichOne);
    }
})
```

SEE ALSO:

- Component IDs
- Finding Components by ID

Using Java

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

IN THIS SECTION:

Essential Terminology
When you write Java code in Aura, it's essential to understand some basic concepts of the framework.

Reading Initial Component Data with Models
A model is a component's main source for dynamic data.

Creating Server-Side Logic with Controllers
The framework supports client-side and server-side controllers. 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 Rendering to the DOM
The Aura rendering service takes in-memory component state and updates the component in the Document Object Model (DOM).

Server-Side Runtime Binding of Components
A provider enables you to use an abstract component in markup. The framework uses the provider to determine the concrete component to use at runtime.

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.

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	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. A definition's metadata can include a name, location of origin, and descriptor (DefDescriptor, the primary key of the definition).
DefDescriptor	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 `model="java://myPackage.MyClass"` with methods to retrieve the

Term	Description
	language, class name, and package name. Rather than passing a more heavyweight definition around in code, Aura usually passes around a DefDescriptor instead. The qualified name for a DefDescriptor has a format of either `prefix://namespace:name` or `prefix://namespace.name`. For example, `js://ui.button`. • prefix: Defines the language, such as JavaScript or Java • namespace: Corresponds to the package name or XML namespace • name: Corresponds to the class name or local name
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 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
{
    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.get("v.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.

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:

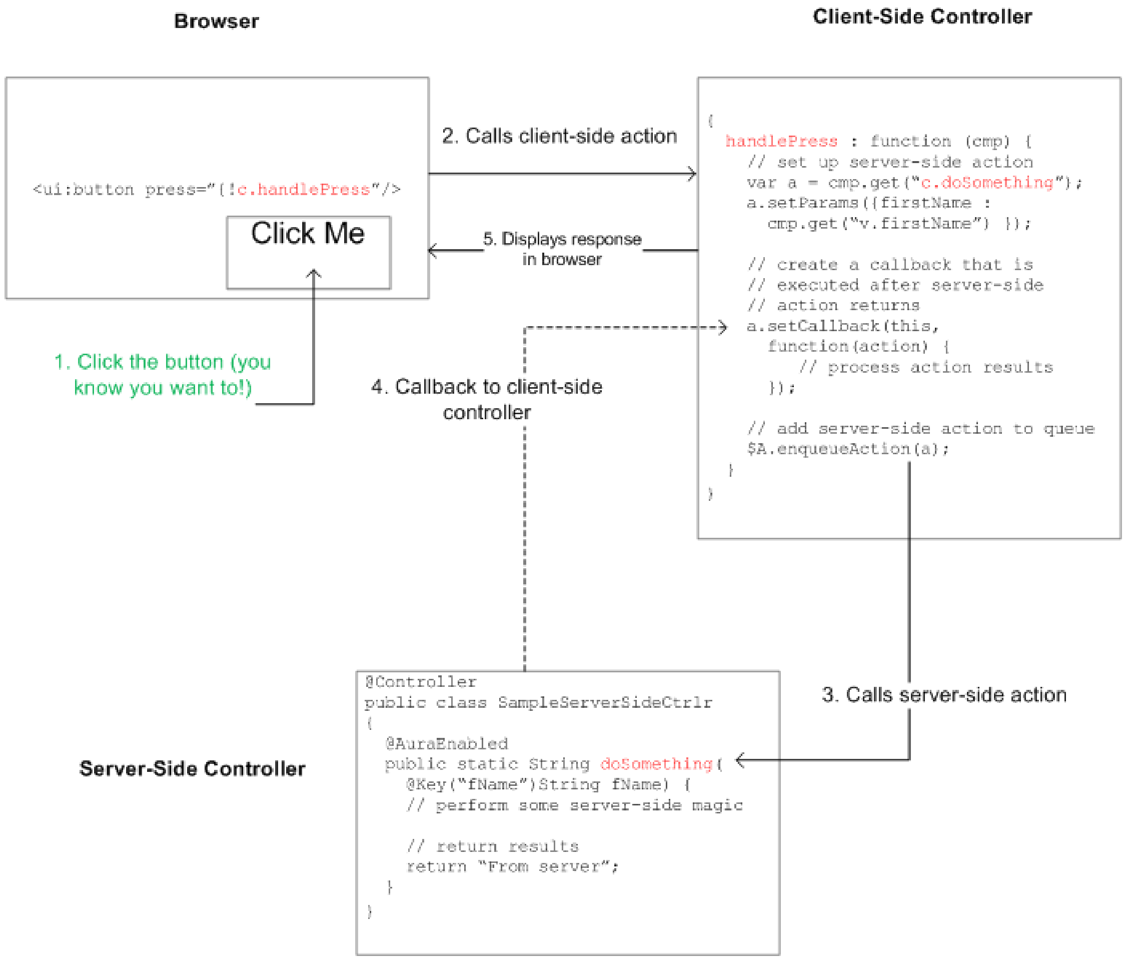
- Java Models
- JSON Models
- Working with Attribute Values in JavaScript

Creating Server-Side Logic with Controllers

The framework supports client-side and server-side controllers. 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.


Browser
Client-Side Controller
2. Calls client-side action
Click Me
5. Displays response in browser
1. Click the button (you know you want to!)
4. Callback to client-side controller
3. Calls server-side action
Server-Side Controller

```
<ui:button press="{!c.handlePress"/>
```

```
{
 handlePress : function (cmp) {
   // set up server-side action
   var a = cmp.get("c.doSomething");
   a.setParams({firstName :
     cmp.get("v.firstName") });

   // create a callback that is
   // executed after server-side
   // action returns
   a.setCallback(this,
     function(action) {
       // process action results
     });

   // add server-side action to queue
   $A.enqueueAction(a);
 }
}
```

```
@Controller
public class SampleServerSideCtrlr
{
  @AuraEnabled
  public static String doSomething(
    @Key("fName")String fName) {
    // perform some server-side magic

    // return results
    return "From server";
  }
}
```

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 180.

IN THIS SECTION:

Creating a Java Server-Side Controller
Create a server-side controller in Java. A component must include a `controller` attribute that wires it to the server-side Java 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 but it enables the framework to minimize network traffic by batching multiple actions into one request.

Foreground and Background Actions

Foreground actions are the default. An action can be marked as a background action. This is useful when you want your app to remain responsive to a user while it executes a low priority, long-running action. A rough guideline is to use a background action if it takes more than five seconds for the response to return from the server.

Abortable Actions

Mark an action as abortable to make it potentially abortable while it's queued to be sent to the server. An abortable action in the queue is not sent to the server if the component that created the action is no longer valid, that is `cmp.isValid() == false`. A component is automatically destroyed and marked invalid by the framework when it is unrendered.

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

Mark an action as storable to have its response stored in the client-side cache by the framework. Caching 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. A component must include a `controller` attribute that wires it to the server-side Java controller.

Here's a sample Java controller that 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.

Tip: Don't store component state in your controller. Store it in a component's attribute or model instead.

```
package org.auraframework.demo.controllers;

@ServiceComponent
public class SimpleServerSideController implements Controller
{

    //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.

`@ServiceComponent`

Denotes that a Java class is a server-side controller. The class must implement the `Controller` interface too.

`@AuraEnabled`

Enables client- and server-side access to a controller method. This means that you only expose methods that you have explicitly annotated.

@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. The framework 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.SimpleServerSideController">
```

SEE ALSO:

Foreground and Background Actions

Component Markup

Calling a Server-Side Action

Call a server-side controller action from a client-side controller. In the client-side controller, you set a callback, which is called after the server-side action is completed. A server-side action can return any object containing serializable JSON data.

A client-side controller is a JavaScript object in object-literal notation containing name-value pairs. Each name corresponds to a client-side action. Its value is the function code associated with the action.

Let's say that you want to trigger a server-call from a component. The following component contains a button that's wired to a client-side controller `echo` action. `SimpleServerSideController` contains a method that returns a string passed in from the client-side controller.

```
<aura:component
controller="java://org.auraframework.demo.controllers.SimpleServerSideController">
    <aura:attribute name="firstName" type="String" default="world"/>
    <ui:button label="Call server" press="{!c.echo}"/>
</aura:component>
```

This client-side controller includes an `echo` action that executes a `serverEcho` method on a server-side controller.

Tip: Use unique names for client-side and server-side actions in a component. A JavaScript function (client-side action) with the same name as a server-side action (Java method) can lead to hard-to-debug issues.

```
({
    "echo" : function(cmp) {
        // create a one-time use instance of the serverEcho action
        // in the server-side controller
        var action = cmp.get("c.serverEcho");
        action.setParams({ firstName : cmp.get("v.firstName") });

        // Create a callback that is executed after
        // the server-side action returns
        action.setCallback(this, function(response) {
            var state = response.getState();
            // This callback doesn’t reference cmp. If it did,
            // you should run an isValid() check
            //if (cmp.isValid() && state === "SUCCESS") {
            if (state === "SUCCESS") {
                // Alert the user with the value returned
                // from the server
                alert("From server: " + response.getReturnValue());

                // You would typically fire a event here to trigger
                // client-side notification that the server-side
                // action is complete
            }
            //else if (cmp.isValid() && state === "INCOMPLETE") {
            else if (state === "INCOMPLETE") {
                // do something
            }
            //else if (cmp.isValid() && state === "ERROR") {
            else if (state === "ERROR") {
                var errors = response.getError();
                if (errors) {
                    if (errors[0] && errors[0].message) {
                        console.log("Error message: " +
                                 errors[0].message);
                    }
                } else {
                    console.log("Unknown error");
                }
            }
        });

        // optionally set storable, abortable, background flag here

        // A client-side action could cause multiple events,
        // which could trigger other events and
        // other server-side action calls.
        // $A.enqueueAction adds the server-side action to the queue.
        $A.enqueueAction(action);
    }
})
```

In the client-side controller, we use the value provider of `c` to invoke a server-side controller action. We also use the `c` syntax in markup to invoke a client-side controller action.

The `cmp.get("c.serverEcho")` call indicates that we're calling the `serverEcho` method in the server-side controller. The method name in the server-side controller must match everything after the `c.` in the client-side call. In this case, that's `serverEcho`.

Use `action.setParams()` to set arguments to be passed to the server-side controller. The following call sets the value of the `firstName` argument on the server-side controller's `serverEcho` method based on the `firstName` attribute value.

```
action.setParams({ firstName : cmp.get("v.firstName") });
```

`action.setCallback()` sets a callback action that is invoked after the server-side action returns.

```
action.setCallback(this, function(response) { ... });
```

The server-side action results are available in the `response` variable, which is the argument of the callback.

`response.getState()` gets the state of the action returned from the server.

Note: Always add an `isValid()` check if you reference a component in asynchronous code, such as a callback or a timeout. If you navigate elsewhere in the UI while asynchronous code is executing, the framework unrenders and destroys the component that made the asynchronous request. You can still have a reference to that component, but it is no longer valid. Add an `isValid()` call to check that the component is still valid before processing the results of the asynchronous request.

`response.getReturnValue()` gets the value returned from the server. In this example, the callback function alerts the user with the value returned from the server.

`$A.enqueueAction(action)` adds the server-side controller action to the queue of actions to be executed. All actions that are enqueued will run at the end of the event loop. Rather than sending a separate request for each individual action, the framework processes the event chain and batches the actions in the queue into one request. The actions are asynchronous and have callbacks. The `runAfter` method is deprecated.

Tip: If your action is not executing, make sure that you're not executing code outside the framework's normal rerendering lifecycle. For example, if you use `window.setTimeout()` in an event handler to execute some logic after a time delay, wrap your code in `$A.getCallback()`. You don't need to use `$A.getCallback()` if your code is executed as part of the framework's call stack; for example, your code is handling an event or in the callback for a server-side controller action.

Action States

The possible action states are:

NEW
The action was created but is not in progress yet

RUNNING
The action is in progress

SUCCESS
The action executed successfully

ERROR
The server returned an error

INCOMPLETE
The server didn't return a response. The server might be down or the client might be offline. The framework guarantees that an action's callback is always invoked as long as the component is valid. If the socket to the server is never successfully opened, or closes abruptly, or any other network error occurs, the XHR resolves and the callback is invoked with state equal to `INCOMPLETE`.

ABORTED

The action was aborted. This action state is deprecated. A callback for an aborted action is never executed so you can't do anything to handle this state.

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 but it enables the framework to minimize network traffic by batching multiple actions into one request.

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.

Tip: If your action is not executing, make sure that you're not executing code outside the framework's normal rerendering lifecycle. For example, if you use `window.setTimeout()` in an event handler to execute some logic after a time delay, wrap your code in `$A.getCallback()`.

There are some properties that you can set on an action to influence how the framework manages the action while it's in the queue waiting to be sent to the server. For more information, see:

- Foreground and Background Actions on page 183
- Abortable Actions on page 184
- Caboose Actions on page 185
- Storable Actions on page 186

SEE ALSO:

Modifying Components Outside the Framework Lifecycle

Foreground and Background Actions

Foreground actions are the default. An action can be marked as a background action. This is useful when you want your app to remain responsive to a user while it executes a low priority, long-running action. A rough guideline is to use a background action if it takes more than five seconds for the response to return from the server.

Batching of Actions

Multiple queued foreground actions are batched in a single request (XHR) to minimize network traffic. Foreground actions are sent in the order that they are received but the server may receive or return the responses in a different order depending on processing time.

Each background action is sent in its own request in the order that it's received. The server responses may return in a different order depending on the processing time of the actions.

When the server-side actions in the queue are executed, the foreground actions execute first and then the background actions execute. Background actions run in parallel with foreground actions and responses of foreground and background actions may come back in either order.

Framework-Managed Request Throttling

The framework throttles foreground and background requests separately. This means that the framework can control the number of foreground requests and the number of background actions running at any time. The framework automatically throttles requests and it's not user controlled. The framework manages the number of foreground and background XHRs, which varies depending on available resources.

Even with separate throttling, background actions might affect performance in some conditions, such as an excessive number of requests to the server.

Setting Background Actions

To set an action as a background action, call the `setBackground()` method on the action object in JavaScript.

```
// set up the server-action action
var action = cmp.get("c.serverEcho");
// optionally set actions params
//action.setParams({ firstName : cmp.get("v.firstName") });
// set as a background action
action.setBackground();
```

Note: 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.

SEE ALSO:

Queueing of Server-Side Actions

Calling a Server-Side Action

Creating a Java Server-Side Controller

Abortable Actions

Mark an action as abortable to make it potentially abortable while it's queued to be sent to the server. An abortable action in the queue is not sent to the server if the component that created the action is no longer valid, that is `cmp.isValid() == false`. A component is automatically destroyed and marked invalid by the framework when it is unrendered.

Note: We recommend that you only use abortable actions for read-only operations as they are not guaranteed to be sent to the server.

An abortable action is sent to the server and executed normally unless the component that created the action is invalid before the action is sent to the server.

A non-abortable action is always sent to the server and can't be aborted in the queue.

If an action response returns from the server and the associated component is now invalid, the logic has been executed on the server but the action callback isn't executed. This is true whether or not the action is marked as abortable.

Marking an Action as Abortable

Mark a server-side action as abortable by using the `setAbortable()` method on the `Action` object in JavaScript. For example:

```
var action = cmp.get("c.serverEcho");
action.setAbortable();
```

SEE ALSO:

Creating Server-Side Logic with Controllers
Queueing of Server-Side Actions
Calling a Server-Side Action

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 waits 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.

If no other actions trigger an XHR to be sent to the server within 60 seconds, any pending caboose actions are batched into their own XHR. The 60-seconds countdown starts when a caboose action is enqueued. The caboose action is sent the next time the framework processes any events after the countdown elapses.

Note: If there is a caboose action in the queue when a user closes the app, that caboose action will not be sent.

Marking Caboose Actions

When you generate data on the client that you want to send to the server, mark a foreground action as a caboose action with `action.setCaboose()`, set a callback with `setAllAboardCallback()`, and enqueue the action using `$A.enqueueAction()`. The `setAllAboardCallback()` callback is called just before the action is sent to the server, just like an "all aboard" announcement before a train leaves a station.

To implement a log and flush pattern, the callback should use one or more calls to `setParam()` on the action to set the data to be sent. The server-side action should then process the data that was sent as parameters.

This sample code in a helper adds log data to a data queue. The caboose action contains the log data and flushes the client-side data queue just before the action is sent to the server.

```
({
    initFields : function(component) {
        /**
         * A queue of log data objects
         */
        this.dataQueue = [];
    },

    /**
     * Add log data to the data queue
     *
     * @param {!string} key App analytics handler key
     * @param {!Object} data App analytics data
     */
    doCaboose : function(key, data) {
        // if data queue is empty, set up caboose action
        if (this.dataQueue.length == 0) {
            // set server-side action
            // serverHandle is a method in the server-side controller
            // that processes the data. The server-side code is not shown here.
            var action = component.get('c.serverHandle');
            action.setAllAboardCallback(this,
                    this.flushDataQueue);
            action.setCaboose();
            $A.enqueueAction(action);
        }
        var logData = {};
        logData[key] = data;
        this.dataQueue.push(logData);
    },

    /**
     * Send the queue to the server and then reset the queue
     *
     * @param {!Object} action Caboose action that is about to be sent to the server
     */
    flushDataQueue : function(action) {
        var batchedData = this.dataQueue;
        this.dataQueue = [];
        action.setParam('batch', batchedData);
    }
})
```

SEE ALSO:

Queueing of Server-Side Actions

Calling a Server-Side Action

Storable Actions

Mark an action as storable to have its response stored in the client-side cache by the framework. Caching 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. Never mark as storable an action that updates or deletes data.

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 framework-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.

Warning: For storable actions in the cache, the framework returns the cached response immediately and also tries to refresh the data for quick display the next time it's requested. Therefore, storable actions might have their callbacks invoked more than once: first with cached data, then with updated data from the server.

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.

Enable Storable Actions for Apps

Storage for server-side actions caches action response values. The storage name must be `actions`.

For an example of initializing storage for an app, see Initializing Storage Service.

Marking Storable Actions

To mark a server-side action as storable, call `setStorable()` on the action in JavaScript code, as follows.

```
action.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 only set the following property:

`ignoreExisting`
Set to `true` to refresh the stored item with a newly retrieved value, regardless of whether the item has expired or not. The default value is `false`.

To set the storage options for the action response, pass this configuration map into `setStorable()`.

Refreshing an Action Response for Every Request

To ignore existing stored responses, set:

```
action.setStorable({
    "ignoreExisting": "true"
});
```

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 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 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

At runtime, a provider has access to a shell of the abstract component, 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 that implement an interface. Aura doesn't currently enforce this restriction, but it's the preferred pattern. The abstract component that references the provider also extends the base component.

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

Java Cookbook

This section includes code snippets and samples that can be used in JavaScript classes.

IN THIS SECTION:

Dynamically Creating Components in Java

You can create a component dynamically in your Java code.

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.

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.

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. For example, the `aura:if` component uses `ComponentDefRef` for its `body` and `else` attributes.

```
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.

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`.

Note: Although `ComponentDefRef` provides performance benefits, we recommend you to use `AuraComponent[]` instead as `ComponentDefRef` will be deprecated in a later release. During component creation, any items marked as an `AuraComponent[]` type is recursively created. Items that are marked as a `ComponentDefRef` is initialized as a list that contains only the information to create the actual components at a later time. For more information, see Component Request Lifecycle.

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

Controlling Access

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

Use the `access` system attribute on these tags:

- `<aura:application>`
- `<aura:attribute>`

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

Access Values

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

Note: If you're an internal Salesforce developer, look at our internal doc as access levels work differently than they do for open-source implementations.

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

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

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

`public`
Available within the same namespace.

`internal`
Available within any internal namespace. An internal namespace is a namespace loaded from the filesystem. This is the default access value.

Resources, such as components, in an internal namespace may access `global`, `internal`, `public` (in the same namespace), or `private` (in the same component) resources.

`global`
Available in all namespaces.

Example

This sample component has global access.

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

Access Violations

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

- Client-side code doesn't execute or returns `undefined`. You also see an error message in your browser console unless you're running in `PROD` mode.
- Server-side code results in the component failing to load. You also see a popup error message unless you're running in `PROD` mode.

Anatomy of an Access Check Error Message

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

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

An error message has four parts:

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

Fixing Access Check Errors

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

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

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

Example: `is not visible to 'undefined'`

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

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

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

Example: `is not visible to 'InvalidComponent ...'`

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

The key word in this error message is `InvalidComponent`, which indicates that `c:sourceComponent` is invalid and has been destroyed.

Always add an `isValid()` check if you reference a component in asynchronous code, such as a callback or a timeout. If you navigate elsewhere in the UI while asynchronous code is executing, the framework unrenders and destroys the component that made the asynchronous request. You can still have a reference to that component, but it is no longer valid. Add an `isValid()` call to check that the component is still valid before processing the results of the asynchronous request.

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

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

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

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

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

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

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

IN THIS SECTION:

Application Access Control

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

Possible values are listed below.

Modifier	Description
`public`	Available within the same namespace.
`internal`	Available within any internal namespace. An internal namespace is a namespace loaded from the filesystem. This is the default access value. If set to `internal`, the app isn't directly accessible via a URL in `PROD` mode.

Modifier	Description
`global`	Available in all namespaces.

Interface Access Control

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

Possible values are listed below.

Modifier	Description
`public`	Available within the same namespace.
`internal`	Available within any internal namespace. An internal namespace is a namespace loaded from the filesystem. This is the default access value.
`global`	Available in all namespaces.

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 controls whether the component can be used outside of the component's namespace.

Possible values are listed below.

Modifier	Description
`public`	Available within the same namespace.
`internal`	Available within any internal namespace. An internal namespace is a namespace loaded from the filesystem. This is the default access value. If set to `internal`, the component isn't directly accessible via a URL in `PROD` mode.
`global`	Available in all namespaces.

Attribute Access Control

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

Possible values are listed below.

Access	Description
`private`	Available within the component, app, interface, event, or method and can't be referenced outside the file. Note: Accessing a private attribute returns `undefined` unless you reference it from the component in which it's declared. You can't access a private attribute from a sub-component that extends the component containing the private attribute.
`public`	Available within the same namespace.
`internal`	Available within any internal namespace. An internal namespace is a namespace loaded from the filesystem. This is the default access value.
`global`	Available in all namespaces.

Event Access Control

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

Possible values are listed below.

Modifier	Description
`public`	Available within the same namespace.
`internal`	Available within any internal namespace. An internal namespace is a namespace loaded from the filesystem. This is the default access value.
`global`	Available in all namespaces.

URL-Centric Navigation

It's useful to understand how the framework handles page requests. The initial GET request for an app retrieves a template containing all the framework 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

The framework uses its event model to manage content change in response to URL changes. The framework monitors the location of the current window for changes. If the `#` value in a URL changes, the framework fires an 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`.

IN THIS SECTION:

SEE ALSO:

Modes Reference

aura:application

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 the framework 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 the framework 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 framework 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 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 Object-Oriented Development

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

For example, components are encapsulated and their internals stay private. Consumers of the component can access the public shape (attributes and registered events) of the component, but can't access other implementation details in the component bundle. This 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.

What is Inherited?

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

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

Component Attributes

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

Events

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

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

When an event fires, handlers for the event are executed. Handlers for any event that extend the event are also executed.

Helpers

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

Controllers

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

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

Models Are Not Inherited

A component's model is **not** inherited by a component that extends a super component.

SEE ALSO:

- Component Attributes
- Communicating with Events
- Sharing JavaScript Code in a Component Bundle
- Handling Events with Client-Side Controllers
- aura:set
- Java Models

Inherited Component Attributes

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

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

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

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

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

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

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

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

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

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

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

Inherited `body` Attribute

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

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

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

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

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

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

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

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

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

`c:subBody` outputs:

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

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

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

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

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

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

SEE ALSO:

- aura:set
- Component Body
- Component Markup

Abstract Components

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

Similarly, 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.

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

- `<aura:attribute>` tags to define the interface's attributes.
- `<aura:registerEvent>` tags to define the events that it may fire.

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

To use an interface, you must implement it. 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 super component. This usage works for components and abstract components, but it doesn't work for interfaces. To set the value of an attribute inherited from an interface, redefine the attribute in the sub component using `<aura:attribute>` and set the value in its default attribute.

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.

SEE ALSO:

Server-Side Runtime Binding of Components

Setting Attributes Inherited from an Interface

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.

Inheritance Rules

This table describes the inheritance rules for various elements.

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

SEE ALSO:

Interfaces

Communicating with Events

Caching with Storage Service

The Storage Service provides a powerful, simple-to-use caching infrastructure that enhances the user experience on the client. 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. Caching is especially beneficial for high-performance, mostly connected applications operating over high latency connections, such as 3G networks.

The Storage Service supports several implementations of storage and selects one at runtime based on browser support and specified characteristics of persistence and security. Storage can be persistent and secure. With persistent storage, cached data is preserved between user sessions in the browser. With secure storage, cached data is encrypted.

Storage Adapter Name	Persistent	Secure
SmartStore	`true`	`true`
IndexedDB	`true`	`false`
MemoryAdapter	`false`	`true`

SmartStore

(Persistent and secure) Provides a caching service that is only available for apps built with 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.

IndexedDB

(Persistent but not secure) Provides access to an API for client-side storage and search of structured data. For more information, see the Indexed Database API.

MemoryAdapter

(Not persistent but secure) Provides access to JavaScript memory for caching data. The stored cache persists only per browser page. Browsing to a new page resets the cache.

The Storage Service selects a storage adapter on your behalf that matches the persistent and secure options you specify when initializing the service. For example, if you request a persistent and insecure storage service, the Storage Service returns the IndexedDB storage if the browser supports it.

To ensure data security, if you request secure storage, the Storage Service always selects a secure adapter even if it doesn't satisfy the request for persistence. For example, persistent and secure storage is only available for custom apps built with the Salesforce Mobile SDK. If you request persistent and secure storage for an app that's not built with the Salesforce Mobile SDK, the Storage Service returns the MemoryAdapter storage, which is secure but not persistent.

The storage name is required and must be unique.

There are two types of storage:

- Server-side actions storage: Storage for server-side actions that enables caching action response values. The storage name must be `actions`.
- Custom named storage: Storage that you control by adding and retrieving items to and from storage. The storage name can be any name except for `actions`, which is reserved for caching action response values.

SEE ALSO:

Creating Server-Side Logic with Controllers

Storable Actions

Initializing Storage Service

Initializing Storage Service

Initialize storage in markup or JavaScript by specifying a name and, optionally, other properties.

If you don't specify the optional properties, the Storage Service uses default values set by the `initStorage()` method of AuraStorageService.

Initialize in Markup

This example uses a template to initialize storage for server-side action response values. The template contains an `<auraStorage:init>` tag that specifies storage initialization properties.

```
<aura:component isTemplate="true" extends="aura:template">
    <aura:set attribute="auraPreInitBlock">
        <!-- Note that the maxSize attribute in <auraStorage:init> is in KB -->
        <auraStorage:init
          name="actions"
```

```
            persistent="false"
            secure="true"
            maxSize="1024" />
    </aura:set>
</aura:component>
```

When you initialize storage, you can set certain options, such as the name, maximum cache size, and the default expiration time.

Storage for server-side actions caches action response values. The storage name must be `actions`.

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.

Initialize in JavaScript

Initialize storage dynamically using the JavaScript API. This example shows how to initialize the Storage Service using `initStorage()` in a JavaScript client-side controller.

```
var storage = $A.storageService.initStorage({
    "name":                 "MyStorage",
    "persistent":           true,
    "secure":               true,
    "maxSize":              524288, // (bytes) (512 * 1024)
    "expiration":           600,    // (seconds)
    "autoRefreshInterval": 600,    // (seconds)
    "debugLogging":         true,
    "clearOnInit":          false,
    "version":              "1.0"
});
```

Warning: The `maxSize` property in `$A.storageService.initStorage()` has a unit of bytes. This is different than the `maxSize` attribute in `<auraStorage:init>`, which has a unit of KB.

Storage Versions

The storage service uses an optional version as part of the key when getting or setting items. This enables you to cache data specific to different versions of your app. You can change the default version for an app. When you retrieve data from the cache using the new version, the cached data for the old version is ignored as it has a different key. This avoids the problem of clients retrieving data associated with an old version from the cache.

There are two types of versions for storage: an app-level default version and a version specific to an individual store. If you don't specify a version when you create storage, the storage inherits the app-level default.

Use `$A.storageService.setVersion()` to create an app-level version. Use the `version` parameter in `$A.storageService.initStorage()` or the `version` attribute in `<auraStorage:init>` to set a storage-specific version when you initialize the storage.

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 `set` and `get` API of AuraStorage.

Storage Service uses ES6 Promises. For more information about promises, see https://developers.google.com/web/fundamentals/getting-started/primers/promises.

`AuraStorage` calls are asynchronous and return a `Promise` object that is resolved when the operation completes, or rejected if an error occurred.

Note: Promises execute their resolve and reject functions asynchronously so the code is outside the Aura event loop and normal rendering lifecycle. If the resolve or reject code makes any calls to Aura, such as setting a component attribute, use `$A.getCallback()` to wrap the code. For more information, see Modifying Components Outside the Framework Lifecycle on page 155.

The framework-provided actions storage for server-side actions automatically adds and retrieves items from storage and doesn't require you to call `set` and `get` explicitly. See Storable Actions on page 186.

Getting and Setting Items

This example shows how to use a storage object to explicitly store items. For information on initializing a storage object, see Initializing Storage Service on page 204.

The call to `set` takes a key that is used to uniquely identify the stored item, and returns a `Promise` that resolves when the operation is complete.

```
var value1 = 67;
// returns a Promise object that is not used here
storage.set("score", value1);

storage.set("name", "joe smith")
  .then(function() { console.log("name is stored"); },
        function(err) { console.log("named failed to store: " + err) }
  );
```

The first function in `then()` is called when the `Promise` resolves. The second function is called when the `Promise` rejects.

You can retrieve stored items by using the `get` method. The `get` method takes as a parameter the key of the object you wish to retrieve. It returns a `Promise` that resolves to the retrieved value or `undefined` if the key is not found.

```
storage.get("score")
  .then(function(value) { console.log("score is " + value); })
  .then(function() { return storage.get("name"); })
  .then(function(value) { console.log("name is " + value); },
        function(err) { console.log("failed: " + err); }
  );
```

Note: If you're getting or setting more than one value, use `getAll()` or `setAll()` for better performance.

Using Other `AuraStorage` Methods

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 available in the JavaScript API. For example, you can get the current and max size:

```
var storage = $A.storageService.getStorage("MyStorage");
storage.getSize().then(
  function(size) { return size; },
  function(err) { return "unknown"; }
).then(function(size) {
  var max = storage.getMaxSize();
  console.log("size is " + size + " KB of max " + max + " KB");
});
```

To clear the storage:

```
var storage = $A.storageService.getStorage("MyStorage";
storage.clear().then(
  function() { console.log("storage has cleared"); },
  function(err) { console.log("storage failed to clear: " + err); }
);
```

SEE ALSO:

Storable Actions

Initializing Storage Service

Using the AppCache

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. The framework supports AppCache for WebKit-based browsers, such as Chrome and Safari.

Note: See an introduction to AppCache for more information.

IN THIS SECTION:

Enabling the AppCache
The framework disables the use of AppCache by default.

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.

Specifying Additional Resources for Caching
When AppCache is enabled, you can specify web resources to be cached in addition to the resources that framework caches by default.

SEE ALSO:

Component Request Overview

aura:application

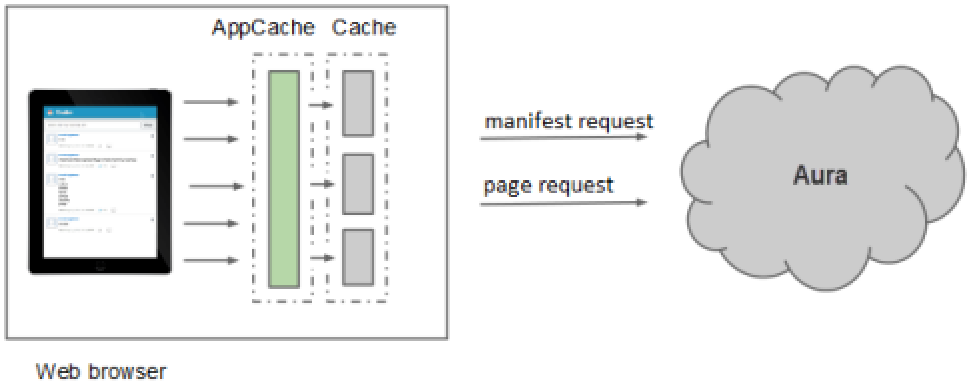
Enabling the AppCache

The framework 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, the framework updates the timestamp to trigger a refetch of all resources. Fetching resources only when necessary reduces server trips for users.

When a browser initially requests an app, a link to the manifest file is included in the response.

```
<html manifest="/path/to/app.manifest">
```

The manifest path includes the mode and app name of the app that's currently running. This manifest file lists framework resources as well as your JavaScript code and CSS, which are cached after they're downloaded for the first time. A hash in the URL ensures that you always have the latest resources.

Note: You'll see different resources depending on which mode you're running in. For example, `aura_prod.js` is available in `PROD` mode and `aura_proddebug.js` is available in `PRODDEBUG` mode.

Specifying Additional Resources for Caching

When AppCache is enabled, you can specify web resources to be cached in addition to the resources that framework 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 6 Testing and Debugging Components

In this chapter ...

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

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>`. Prepending an underscore to the test function name like `_testGetResult` disables the test. 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 **/
    testCase1: {
        attributes: {
            //Attributes
        },
        //Runs all supported browsers except Firefox.
        //Overrides the suite level browsers tag.
        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`
Applies to suite or test level. Setting attribute values outside of a test case applies the attribute values to the whole suite. If a test has both test level and suite level attributes, the test level attributes override those at the suite level.

Attribute values 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'
}
```

auraWarningsExpectedDuringInit

Applies to test level only and accepts an array of Strings, where each String is an expected warning message. When the `failOnWarning` flag is set, `auraWarningsExpectedDuringInit` specify warnings from `$A.warning` allowed during initialization that won't fail the test. For more information, see Fail a Test Only When Expected on page 214.

browsers

Applies to suite or test level. 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' ]
```

By default, the tests run on desktop and mobile browsers. To test only for desktop or mobile browsers, specify the `DESKTOP` or `MOBILE` attribute.

```
browsers: [ 'DESKTOP'] // Run tests for desktop browsers only
```

If the `browsers` property is available in both the case level and the suite level, the case level property overrides the latter. Desktop browsers include `IE9`, `IE10`, `IE11`, `FIREFOX`, `GOOGLECHROME`, and `SAFARI`. Mobile browsers include `IPHONE`, `IPAD`, `ANDROID_PHONE`, `ANDROID_TABLET`.

doNotWrapInAuraRun

Applies to suite or test level. Each function block within a test is referred to as a stage. By default, each stage of the test executes within its own $A.run() function to ensure the test code goes through the full rendering lifecycle and that all enqueued actions are run before continuing on to the next stage or completing the test. If this is not the desired behavior for your tests, set `doNotWrapInAuraRun` to true.

failOnWarning

Applies to suite or test level. If true, the test will fail on any warnings received during test execution not marked as expected via a call to `$A.test.expectWarnings()`, or any warnings received during test setup not declared under the `auraWarningsExpectedDuringInit` tag. For more information, see Fail a Test Only When Expected on page 214.

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 received was null');
    }
```

sharedString

Share a string or function for multiple tests in the same test file.

```
sharedString: "My shared string",
sharedFunction: function(){},
testFunction:function(){
    this.sharedFunction(this.sharedString);
}
```

`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 221.

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. After the first function runs, the test waits for `$A.test.addWaitFor` to complete. This example method compares `expected` and the return value of the `lookForTextAfterClick`. When this comparison evaluates to true, the next function is run.

```
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.addWaitFor('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.');
}]
```

To display an error message when the test function times out, use `$A.test.addWaitForWithFailureMessage`. This example runs a test that expects a result length of one by default, or two if the component is rendered on a phone.

```
test: function(cmp) {
    var expectedResultLength = 1;
    if ($A.get("$Browser.formFactor") == 'PHONE') {
        expectedResultLength = 2;
    }
    $A.test.addWaitForWithFailureMessage(
       expectedResultLength,
       function() {
           return result.length;
       },
       "Unexpected number of items in result");
}
```

IN THIS SECTION:

Pass a Controller Action in Component Tests

Invoke a controller action by wrapping it in a component.

Fail a Test Only When Expected
You can set a test to expect an error or warning, and fail a test only when you expect it to fail.

Pass a Controller Action in Component Tests

Invoke a controller action by wrapping it in a component.

You can't pass in a function or action as an attribute in component tests. Instead, use a component wrapper. For example, you want to pass an action in the following component to a test.

```
<!-- myNamespace:childCmp -->
<aura:component>
    <aura:attribute name="put" type="Aura.Action"/>
</aura:component>
```

Use a component wrapper that looks like this:

```
<aura:component>
    <aura:attribute name="items" type="integer" default="0"/>
    box called {!v.items} times
    <myNamespace:childCmp aura:id="putter" put="{!c.box}"/>
</aura:component>
```

The controller action retrieves the items attribute in the component and increments its value.

```
({
    box: function(cmp, event) {
        var items = cmp.get("v.items");
        cmp.set("v.items", items + 1);
    }
})
```

The following test verifies the type of action that's passed in the component. `auraType` checks if the object is of a certain type, for example, whether it's of type `Component`, `Action`, or `Event`.

```
testAction: {
    test: function(component){
        var box = component.get("c.box");
        $A.test.assertAuraType("Action", box, "The type was incorrect.");
        $A.test.assertEquals("function", typeof box.run, "The run was not a function on
the actions.");
    }
}
```

Note: The `auraType` attribute is deprecated. Use `$A.test.assertAuraType` to check if a value is an instance of the expected type.

Fail a Test Only When Expected

You can set a test to expect an error or warning, and fail a test only when you expect it to fail.

All tests will fail on errors by default. There is no tag/setting to make a test not fail on errors. You must mark each individual error as expected. All tests will pass on warnings by default. If `failOnWarning: true` is set, then the test will fail for any warnings not marked as expected.

To enable your test to expect an error or warning during initialization, use `auraErrorsExpectedDuringInit` or `auraWarningsExpectedDuringInit`. The test fails only if any of the expected errors don't happen. To enable your test to expect an error or warning during the test itself, use `$A.test.expectAuraError` or `$A.test.expectAuraWarning`.

```
({
    failOnWarning: true,

    /**
     * This case inherits the suite level failOnWarning so we need to declare
auraWarningsExpectedDuringInit to account
     * for warning in the controller's init function and have $A.test.expectAuraWarning
for any warnings in the test
     * block itself.
     */
    testExpectedWarning: {
        auraWarningsExpectedDuringInit: ["Expected warning from auraWarningTestController
 init"],
        test: function(cmp) {
            var warningMsg = "Expected warning from testExpectedWarning";
            var warningMsg2 = "Expected warning from testExpectedWarning2";
            $A.test.expectAuraWarning(warningMsg);
            $A.test.expectAuraWarning(warningMsg2);
            $A.warning(warningMsg);
            $A.warning(warningMsg2);
        }
    },

    /**
     * Override suite level failOnWarning and verify test does not fail on warnings.
     */
    testNoFailOnWarning: {
        failOnWarning: false,
        test: function(cmp) {
            $A.warning("Expected warning from testNoFailOnWarning");
        }
    }
})
```

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 `$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
`$A.test.assert(condition, assertMessage)`	Asserts that the condition is `true`.
`$A.test.assertAccessible(errorMessage)`	Asserts that the HTML output of the target component is accessibility compliant.

Assertion	Description
`$A.test.assertAuraType(type, condition, assertMessage)`	Asserts that the value is an instance of the expected type. Valid types: • Action • ActionDef • Event • EventDef • Component • ComponentDef • ControllerDef • HelperDef • RendererDef • ProviderDef • ModelDef
`$A.test.assertDefined(arg1, assertMessage)`	Asserts that `arg1` is defined.
`$A.test.assertEquals(arg1, arg2, assertMessage)`	Asserts that `arg1 === arg2` is `true`, where `arg1` is the expected value and `arg2` is the actual value.

`$A.test.fail(assertMessage)`	Throws an error with the `assertMessage` string. Use this to test error handling. For example:
	`try {`
	`    // do something where you expect an error`
	`    $A.test.fail("should have got an error");`
	`}`
	`catch(e){`
	`    // assert expected error`
	`}`

`$A.test.assertFalse(condition, assertMessage)`	Asserts that the condition is `false`.
`$A.test.assertFalsy(condition, assertMessage)`	Asserts that the condition is `false`, `null`, or `undefined`.
`$A.test.assertNotEquals(arg1, arg2, assertMessage)`	Asserts that `arg1 === arg2` is `false`, where `arg1` is the expected value and `arg2` is the actual value..
`$A.test.assertNull(arg1, assertMessage)`	Asserts that `arg1` is `null`. If it's not `null`, throws an error with the `assertMessage` string.
`$A.test.assertStartsWith(start, full, assertMessage)`	Asserts that the `full` string starts with the `start` string.
`$A.test.assertNotNull(arg1, assertMessage)`	Asserts that `arg1` is not `null`.

Assertion	Description
`$A.test.assertTrue(condition, assertMessage)`	Asserts that the condition is `true`. This is the same as `$A.test.assert(condition, assertMessage)`.
`$A.test.assertTruthy(condition, assertMessage)`	Asserts that the condition is `true`, `null`, or defined.
`$A.test.assertUndefined(arg1, assertMessage)`	Asserts that the argument is undefined.
`$A.test.assertUndefinedOrNull(arg1, assertMessage)`	Asserts that the argument is undefined or null.
`$A.test.assertNotUndefinedOrNull(arg1, assertMessage)`	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: For a full list of assertions, refer to the JavaScript API reference on page 259.

SEE ALSO:

Supporting Accessibility

Debugging Components

Use the `debugger;` statement to debug your JavaScript tests, with the debug console in your browser opened. 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.

To simulate a user interaction in a test case, fire the associated Aura event. For example, use `buttonComponent.get("e.press").fire()` to simulate a button click event. To fire this event in the browser console, use `$A.getRoot().find("buttonId").get("e.press").fire()`. `$A.getRoot()` returns a reference to the top level component. `buttonId` refers to the local ID of the button component.

SEE ALSO:

Debugging

Communicating with Events

Modes Reference

Utility Functions

Utility functions provides additional support for Aura's unit testing and should be prefixed with `$A.test`.

Utility	Description
`$A.test.addFunctionHandler(instance, originalFunction, newFunction, postProcess)`	Adds a new function handler and overrides the original function.
`$A.test.addPrePostSendCallback(action, preSendCallback, postSendCallback)`	Inserts a callback either before or after sending of XHR. One of `preSendCallback` or `postSendCallback` can be null, but not both.
`$A.test.addWaitFor(expected, testFunction, callback)`	Waits for `expected === testFunction()`.
`$A.test.blockRequests()`	Blocks requests (actions) from being sent to the server.
`$A.test.callServerAction(action, doImmediate)`	Runs a server action. The test waits for any actions to complete before running the next function. If `doImmediate` is set to true, the request is sent immediately. Otherwise, the action is queued after prior requests.
`$A.test.clickOrTouch(element, canBubble, cancelable)`	Fires a touchstart and touchend event if an element supports touch events. Fires a click event on the element otherwise.
`$A.test.expectAuraWarning(msg)`	Tells the test that a warning is expected from `$A.warning` containing `msg` during test execution. The test fails when an unexpected warning is received, if the `failOnWarnings` flag is set for the test.
`$A.test.getErrors()`	Returns errors as JSON encoded strings. If no errors are found, return an empty string.
`$A.test.getExternalAction(component, descriptor, params, returnType, callback)`	Returns an instance of a server action that's unavailable to the component.
`$A.test.getOuterHtml(node)`	Returns the outer HTML of an element.
`$A.test.getPrototype(instance)`	Returns the prototype of the instance or object.
`$A.test.getAction(component, name, params, callback)`	Returns an instance of an action.
`$A.test.getText(node)`	Returns text as a string.
`$A.test.isComplete()`	Returns whether the test is finished running.
`$A.test.overrideFunction(instance, originalFunction, newFunction)`	Overrides an existing function.
`$A.test.print(value)`	Returns the `value` cast to a string. Possible return values are: • undefined

Utility	Description
	• null • `"value"`—the `value` cast to a string • `value.toString()`—for non-strings
`$A.test.releaseRequests()`	Release requests (actions) to be sent to the server.
`$A.test.runAfterIf(conditionFunction, callback, intervalInMs)`	Evaluates `conditionFunction` every interval. When it returns a truthy value, execute the callback. `intervalInMs` is 500 milliseconds by default. Note: Most values in JavaScript are truthy, such as objects, arrays, non-zero numbers, and non-empty strings.
`$A.test.select()`	Returns a list of elements within the document that matches the given arguments.
`$A.test.setTestTimeout(timeoutMsec)`	Sets the timeout in milliseconds from now.

Note: For a full list of utility methods and arguments, refer to the JavaScript API reference on page 259.

SEE ALSO:

Assertions

Sample Test Cases

Testing Label Values

This component contains two link buttons that save or cancel an action.

```
<!-- Component markup -->
<ui:outputURL label="Cancel" value="#" class="secondary-button" linkClick="{!c.onCancel}"/>
<ui:outputURL label="Save" value="#" class="primary-button" linkClick="{!c.onSave}"/>
```

The following test case uses assert statements to check that labels on the link buttons are set correctly. If you're using the global value provider `$Label` to set the label value in the component, use `$A.get("$Label.myLabel")` to retrieve the label.

```
({
browsers: ["-IE7", "-IE8"], //optional browser exclusion
testButtons : {
    test : [
        function testCancelButton(cmp) {
            $A.test.assertEquals(1, $A.test.select('.secondary-button').length,
                    'Cancel button is not being displayed.');

            $A.test.assertEquals("Cancel",
$A.test.getText($A.test.select('.secondary-button')[0]),
                    'Cancel button label is not set correctly.');
},

        function testSaveButton(cmp) {
            $A.test.assertEquals(1, $A.test.select('.primary-button').length,
                    'Save button is not being displayed.');

            $A.test.assertEquals("Save",
$A.test.getText($A.test.select('.primary-button')[0]),
                   'Save button label is not set correctly.');
        }
    ]
}
})
```

Testing Attribute Values

This component contains an input text area component with an attribute that sets the maximum number of characters.

```
<aura:attribute name="maxLength" type="Integer" default="5000"
                description="Max number of chars that can be inserted"/>
<ui:inputTextArea aura:id="textarea" label="My input"/>
```

The following test case checks that the attribute `maxLength` is correctly set.

```
{(
    testMaxLength:{
        attributes : { maxLength: 10 },
        test : function(cmp){
            cmp.set("v.value", "1234567890");
            cmp.getDef().getHelper().onValueChange(cmp);
            $A.test.assertEquals(0, this.getErrorCount(cmp), "No errors found");

            cmp.set("v.value", "12345678901");
            cmp.getDef().getHelper().onValueChange(cmp);
            $A.test.assertEquals(1, this.getErrorCount(cmp), "Too many characters");
    }
}
})
```

Testing HTML Elements

This component contains a `div` tag with a `class` attribute that is set on rendering.

```
<!-- Component Markup -->
<aura:attribute name="class" type="String" default="" description="Additional css classes"/>
<div aura:id="myCmp" class="{! 'myClass ' + v.class}">
    <!-- Other component markup -->
</div>
```

The following test case checks that the specified class is set on initial render and rerender.

```
({
    verifyMyCmp: function(cmp) {
        var ele = cmp.find("myCmp").getElement();
        $A.test.assertTrue($A.util.hasClass(ele, "myClass"), "Element is not rendered with
myClass");

        // additional class
        if(cmp.get("v.class")) {
            $A.test.assertTrue($A.util.hasClass(ele, cmp.get("v.class")), "Additional class
not added as expected");
        }
        else {
            $A.test.assertTrue($A.util.hasClass(ele, "testClass"), "Additional class added
unexpectedly");
        }
    },
    testMyCmp: {
        attributes: {
            isVisible: true, //myCmp is rendered
            'class': "testClass",
        },
        test: function(cmp) {
            this.verifyMyCmp(cmp);
        }
    }
})
```

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",
```

IN THIS SECTION:

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 7 Customizing Behavior with Modes

In this chapter ...

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`
Usage	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.
Debugging	Not recommended for debugging. Since `PROD` mode is intended for apps in production, test modes, such as `SELENIUM`, 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.	Facilitates debugging. JavaScript is non-minified and readable.
Access	Disables access to a `.cmp` resource in a URL. You can only access a `.app` resource.	Enables a `.cmp` resource to be addressed in a URL.	Similar to `PROD` mode
Google Closure Compiler	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
Caching	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 `PROD` 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`	Use for running component tests. If your component or app has a `<componentName>Test.js` 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 `AUTOJSTEST` mode and runs the single test case. The test results are displayed below the iframe. For a successful test run, the tab turns green; for a failure, it turns red.
`JSTESTDEBUG`	Use for debugging component tests. Similar to `JSTEST` mode but doesn't use the Google Closure Compiler.
`AUTOJSTEST`	Used by `JSTEST` mode when running inside the iframe for a test case. It enables extra JavaScript needed to execute the test case. Use this mode by requesting the component or app containing the test in `JSTEST` mode.
`AUTOJSTESTDEBUG`	Used by `JSTESTDEBUG` mode when running inside the iframe for a test case. It enables extra JavaScript needed to execute the test case. Use this mode by requesting the component or app containing the test in `JSTESTDEBUG` mode.
`PTEST`	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. This mode doesn't use the Google Closure Compiler.
`CADENCE`	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. Cadence tests use Jiffy, but don't load the Jiffy Graph UI.
`SELENIUM`	Use for tests with Selenium, a software testing framework for web apps. This mode uses the Google Closure Compiler.
`SELENIUMDEBUG`	Similar to `SELENIUM` 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.

Mode	Usage
`FTEST`	Similar to `UTEST` mode, but used for functional tests instead of unit tests. This mode may expose different debug code than `UTEST` mode.

SEE ALSO:

- Component Bundles
- Setting the Default Mode
- Testing and Debugging 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 8 Debugging

In this chapter ...

There are a few basic tools and techniques that can help you to debug applications.

Use Chrome DevTools to debug your client-side code.

- To open DevTools on Windows and Linux, press Control-Shift-I in your Google Chrome browser. On Mac, press Option-Command-I.
- To quickly find which line of code is failing, enable the **Pause on all exceptions** option before running your code.

To learn more about debugging JavaScript on Google Chrome, refer to the Google Chrome's DevTools website.

Log Messages

To help debug your client-side code, you can write output to the JavaScript console of a web browser using `console.log()` if your browser supports it..

For instructions on using the JavaScript console, refer to the instructions for your web browser.

Use the `$A.log(string[, error])` method to output a log message to the JavaScript console.

The first parameter is the string to log.

The optional second parameter is an error object that can include more detail.

Note: `$A.log()` doesn't output by default in `PROD` or `PRODDEBUG` modes. To log messages in `PROD` or `PRODDEBUG` modes, see Logging in Production Modes on page 232. Alternatively, use `console.log()` if your browser supports it.

For example, `$A.log("This is a log message")` outputs to the JavaScript console:

```
This is a log message
```

Adding `$A.log("The name of the action is: " + this.getDef().getName())` in an action called `openNote` in a client-side controller outputs to the JavaScript console:

```
The name of the action is: openNote
```

The output is also sent to the Aura Debug Tool.

Logging in Production Modes

To log messages in `PROD` or `PRODDEBUG` modes, write a custom logging function. You must use `$A.logger.subscribe(String level, function callback)` to subscribe to log messages at a certain severity level.

The first parameter is the severity level you're subscribing to. The valid values are:

- `ASSERT`
- `ERROR`
- `INFO`
- `WARNING`

The second parameter is the callback function that will be called when a message at the subscribed severity level is logged.

Note that `$A.log()` logs a message at the `INFO` severity level. Adding `$A.logger.subscribe("INFO", logCustom)` causes `$A.log()` to log using the custom `logCustom()` function you define.

Let's look at some sample JavaScript code in a client-side controller.

```
({
    sampleControllerAction: function(cmp) {
        // subscribe to severity levels
        $A.logger.subscribe("INFO", logCustom);
        // Following subscriptions not exercised here but shown for completeness
        //$A.logger.subscribe("WARNING", logCustom);
        //$A.logger.subscribe("ASSERT", logCustom);
        //$A.logger.subscribe("ERROR", logCustom);

        $A.log("log one arg");
        $A.log("log two args", {message: "drat and double drat"});

        function logCustom(level, message, error) {
            console.log(getTimestamp(), "logCustom: ", arguments);
        }

        function getTimestamp() {
            return new Date().toJSON();
        }
    }

})
```

`$A.logger.subscribe("INFO", logCustom)` subscribes so that messages logged at the `INFO` severity level will call the `logCustom()` function. In this case, `logCustom()` simply logs the message to the console with a timestamp.

The `$A.log()` calls log messages at the `INFO` severity level, which matches the subscription and invokes the `logCustom()` callback.

Warning Messages

To help debug your client-side 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.");` outputs to the JavaScript console.

```
This is a warning message.
```

The output is also sent to the Aura Debug Tool.

Note: `$A.warning()` doesn't output by default in `PROD` or `PRODDEBUG` modes. To log warning messages in `PROD` or `PRODDEBUG` modes, use `$A.logger.subscribe("WARNING", logCustom)`, where `logCustom()` is a custom function that you define. For more information, see Logging in Production Modes on page 232.

For instructions on using the JavaScript console, refer to the instructions for your web browser.

CHAPTER 9 Fixing Performance Warnings

In this chapter ...

A few common performance anti-patterns in code prompt the framework to log warning messages to the browser console. Fix the warning messages to speed up your components!

The warnings display in the browser console only if you enabled debug mode.

SEE ALSO:

Enable Debug Mode for Lightning Components

`<aura:if>`—Clean Unrendered Body

This warning occurs when you change the `isTrue` attribute of an `<aura:if>` tag from `true` to `false` in the same rendering cycle. The unrendered body of the `<aura:if>` must be destroyed, which is avoidable work for the framework that slows down rendering time.

Example

This component shows the anti-pattern.

```
<!--c:ifCleanUnrendered-->
<aura:component>
    <aura:attribute name="isVisible" type="boolean" default="true"/>
    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <aura:if isTrue="{!v.isVisible}">
        <p>I am visible</p>
    </aura:if>
</aura:component>
```

Here's the component's client-side controller.

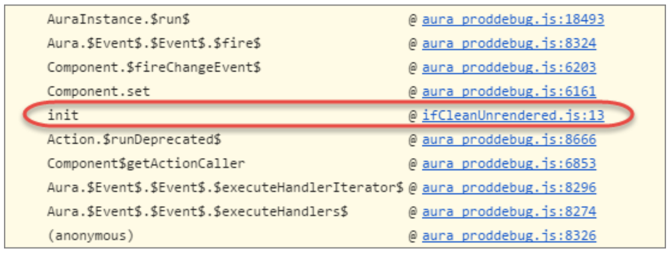
```
/* c:ifCleanUnrenderedController.js */
({
    init: function(cmp) {
        /* Some logic */
        cmp.set("v.isVisible", false); // Performance warning trigger
    }
})
```

When the component is created, the `isTrue` attribute of the `<aura:if>` tag is evaluated. The value of the `isVisible` attribute is `true` by default so the framework creates the body of the `<aura:if>` tag. After the component is created but before rendering, the `init` event is triggered.

The `init()` function in the client-side controller toggles the `isVisible` value from `true` to `false`. The `isTrue` attribute of the `<aura:if>` tag is now `false` so the framework must destroy the body of the `<aura:if>` tag. This warning displays in the browser console only if you enabled debug mode.

```
WARNING: [Performance degradation] markup://aura:if ["5:0"] in c:ifCleanUnrendered ["3:0"]
needed to clear unrendered body.
```

Click the expand button beside the warning to see a stack trace for the warning.



```
AuraInstance.$run$                                  @ aura_proddebug.js:18493
Aura.$Event$.$Event$.$fire$                         @ aura_proddebug.js:8324
Component.$fireChangeEvent$                         @ aura_proddebug.js:6203
Component.set                                       @ aura_proddebug.js:6161
init                                                @ ifCleanUnrendered.js:13
Action.$runDeprecated$                              @ aura_proddebug.js:8666
Component$getActionCaller                           @ aura_proddebug.js:6853
Aura.$Event$.$Event$.$executeHandlerIterator$ @ aura_proddebug.js:8296
Aura.$Event$.$Event$.$executeHandlers$              @ aura_proddebug.js:8274
(anonymous)                                         @ aura_proddebug.js:8326
```

Click the link for the `ifCleanUnrendered` entry in the stack trace to see the offending line of code in the Sources pane of the browser console.

How to Fix the Warning

Reverse the logic for the `isTrue` expression. Instead of setting the `isTrue` attribute to `true` by default, set it to `false`. Set the `isTrue` expression to true in the `init()` method, if needed.

Here's the fixed component:

```
<!--c:ifCleanUnrenderedFixed-->
<aura:component>
    <!-- FIX: Change default to false.
         Update isTrue expression in controller instead. -->
    <aura:attribute name="isVisible" type="boolean" default="false"/>
    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <aura:if isTrue="{!v.isVisible}">
        <p>I am visible</p>
    </aura:if>
</aura:component>
```

Here's the fixed controller:

```
/* c:ifCleanUnrenderedFixedController.js */
({
    init: function(cmp) {
        // Some logic
        // FIX: set isVisible to true if logic criteria met
        cmp.set("v.isVisible", true);
    }
})
```

SEE ALSO:

Enable Debug Mode for Lightning Components

`<aura:iteration>`—Multiple Items Set

This warning occurs when you set the `items` attribute of an `<aura:iteration>` tag multiple times in the same rendering cycle.

There's no easy and performant way to check if two collections are the same in JavaScript. Even if the old value of `items` is the same as the new value, the framework deletes and replaces the previously created body of the `<aura:iteration>` tag.

Example

This component shows the anti-pattern.

```
<!--c:iterationMultipleItemsSet-->
<aura:component>
    <aura:attribute name="groceries" type="List"
                    default="[ 'Eggs', 'Bacon', 'Bread' ]"/>

    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <aura:iteration items="{!v.groceries}" var="item">
        <p>{!item}</p>
    </aura:iteration>
</aura:component>
```

Here's the component's client-side controller.

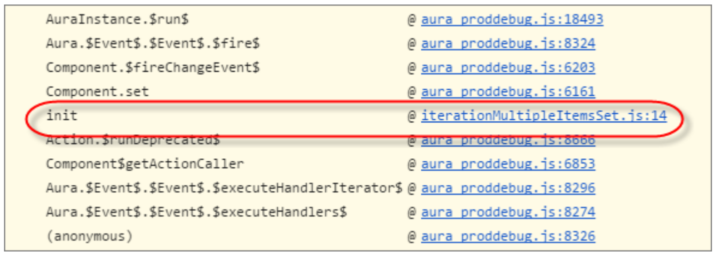
```
/* c:iterationMultipleItemsSetController.js */
({
    init: function(cmp) {
        var list = cmp.get('v.groceries');
        // Some logic
        cmp.set('v.groceries', list); // Performance warning trigger
    }
})
```

When the component is created, the `items` attribute of the `<aura:iteration>` tag is set to the default value of the `groceries` attribute. After the component is created but before rendering, the `init` event is triggered.

The `init()` function in the client-side controller sets the `groceries` attribute, which resets the `items` attribute of the `<aura:iteration>` tag. This warning displays in the browser console only if you enabled debug mode.

```
WARNING: [Performance degradation] markup://aura:iteration [id:5:0] in
c:iterationMultipleItemsSet ["3:0"]
had multiple items set in the same Aura cycle.
```

Click the expand button beside the warning to see a stack trace for the warning.



```
AuraInstance.$run$                               @ aura_proddebug.js:18493
Aura.$Event$.$Event$.$fire$                      @ aura_proddebug.js:8324
Component.$fireChangeEvent$                      @ aura_proddebug.js:6203
Component.set                                    @ aura_proddebug.js:6161
init                                             @ iterationMultipleItemsSet.js:14
Action.$runDeprecated$                           @ aura_proddebug.js:8666
Component$getActionCaller                        @ aura_proddebug.js:6853
Aura.$Event$.$Event$.$executeHandlerIterator$ @ aura_proddebug.js:8296
Aura.$Event$.$Event$.$executeHandlers$           @ aura_proddebug.js:8274
(anonymous)                                      @ aura_proddebug.js:8326
```

Click the link for the `iterationMultipleItemsSet` entry in the stack trace to see the offending line of code in the Sources pane of the browser console.

How to Fix the Warning

Make sure that you don't modify the `items` attribute of an `<aura:iteration>` tag multiple times. The easiest solution is to remove the default value for the `groceries` attribute in the markup. Set the value for the `groceries` attribute in the controller instead.

The alternate solution is to create a second attribute whose only purpose is to store the default value. When you've completed your logic in the controller, set the `groceries` attribute.

Here's the fixed component:

```
<!--c:iterationMultipleItemsSetFixed-->
<aura:component>
    <!-- FIX: Remove the default from the attribute -->
    <aura:attribute name="groceries" type="List" />
    <!-- FIX (ALTERNATE): Create a separate attribute containing the default -->
    <aura:attribute name="groceriesDefault" type="List"
                  default="[ 'Eggs', 'Bacon', 'Bread' ]"/>
    <aura:handler name="init" value="{!this}" action="{!c.init}"/>

    <aura:iteration items="{!v.groceries}" var="item">
        <p>{!item}</p>
    </aura:iteration>
</aura:component>
```

Here's the fixed controller:

```
/* c:iterationMultipleItemsSetFixedController.js */
({
    init: function(cmp) {
        // FIX (ALTERNATE) if need to set default in markup
        // use a different attribute
        // var list = cmp.get('v.groceriesDefault');
        // FIX: Set the value in code
        var list = ['Eggs', 'Bacon', 'Bread'];
        // Some logic
        cmp.set('v.groceries', list);
    }
})
```

SEE ALSO:

Enable Debug Mode for Lightning Components

CHAPTER 10 Measuring Performance with `MetricsService`

In this chapter ...

`MetricsService` enables you to instrument and measure the performance of your code and the framework during development, testing, or production usage. With `MetricsService`, you can abstract your performance marks and measures using plugins. This leads to a clean separation between functional code and instrumentation code that measures the performance of the functional code.

The framework is well instrumented already. You can take advantage of the underlying framework measurements and get insight into the performance of your code by adding a mark or transaction.

Here are some core concepts for the `MetricsService`.

Mark

A mark measures a specific event. Use a mark to measure an interval of a larger transaction.

Transaction

A transaction enables you to track all optional marks that occur in between the transaction start and end time. A mark measures a specific event. A transaction that doesn't contain any marks still tracks useful information about the time taken to complete an operation.

Beacon

A beacon is a component that receives the metrics and sends them somewhere for storage. A beacon abstracts the transport layer for sending collected metrics and transactions.

Plugin

A plugin hooks into the code being measured and enables you to instrument your functional code without adding performance marks directly in your functional code. Add the performance marks in the plugin so that your functional code doesn't get littered with marks. This leads to a clean separation between functional code and instrumentation code that measures the performance of the functional code.

A plugin uses AOP (aspect-oriented programming) and the `MetricsService` API calls to hook into the code being measured.

Adding Performance Transactions

A transaction enables you to track all optional marks that occur in between the transaction start and end time. A mark measures a specific event. A transaction that doesn't contain any marks still tracks useful information about the time taken to complete an operation.

A transaction gives you information about marks that you don't control or own. For example, your transaction could include framework-level marks to track a server request or action caching. This framework-level benchmarking comes for free and can give you valuable insight into the performance of your code.

You can add a transaction directly in your code. Consider adding a transaction when you want to measure an action in production that involves a server trip. This gives you performance data that factors in network latency. You don't need transactions for purely client-side operations as those operations are adequately tested in framework code.

Starting a Transaction

To start a transaction, use `$A.metricsService.transactionStart()`. The syntax is:

```
transactionStart(String ns, String name, Object config)
```

The parameters are:

`String` *`ns`*
Optional. Transaction namespace. You can use any value. This parameter doesn't have anything to do with a component's namespace though you can use the component's namespace as a high-level identifier.

`String` *`name`*
Transaction name. You can use any value, such as a component or action's name.

`Object` *`config`*
Optional custom data to log. Keys in the object are:

`Object` ***`context`***: Custom data for the transaction

`function` ***`postProcess`***: The function to execute before sending the transaction to the beacon

`Boolean` ***`skipPluginPostProcessing`***: If `true`, skip all post processing. This is always set to `true` in `PROD` mode.

Ending a Transaction

To end a transaction, use `$A.metricsService.transactionEnd()`. The syntax is:

```
transactionEnd(String ns, String name, Object config | function)
```

The parameters are:

`String` *`ns`*
Optional. Transaction namespace. You can use any value. This parameter doesn't have anything to do with a component's namespace though you can use the component's namespace as a high-level identifier.

`String` *`name`*
Transaction name. You can use any value, such as a component or action's name.

`Object` *`config`* `|` `function`
Optional. This parameter can be an `Object` or a function. The `Object` contains any custom data that you want to log. If a function is set instead, the function is executed before sending the transaction to the beacon. Keys in the object are:

`Object` ***`context`***: Custom data for the transaction `function` ***postProcess***: The function to execute before sending the transaction to the beacon

`Boolean` ***skipPluginPostProcessing***: If `true`, skip all post processing. This is always set to `true` in `PROD` mode.

Tracking a Specific User Action

To track a specific user action, use `$A.metricsService.transaction()`. Tracking a user taking a specific UI action, such as clicking a specific button, can be useful if you want to analyze these UI actions later. The syntax is:

```
transaction(String ns, String name, Object config | function)
```

The parameters are:

`String` *`ns`*
Optional. Transaction namespace. You can use any value. This parameter doesn't have anything to do with a component's namespace though you can use the component's namespace as a high-level identifier.

`String` *`name`*
Transaction name. You can use any value, such as a component or action's name.

`Object` *`config`* `|` `function`
Optional. This parameter can be an `Object` or a function. The `Object` contains any custom data that you want to log. If a function is set instead, the function is executed before sending the transaction to the beacon. Keys in the object are:

`Object` ***context***: Custom data for the transaction

`function` ***postProcess***: The function to execute before sending the transaction to the beacon

`Boolean` ***skipPluginPostProcessing***: If `true`, skip all post processing. This is always set to `true` in `PROD` mode.

Hook for Callback After Every Transaction Ends

To set a callback to be executed after every transaction ends, use `$A.metricsService.onTransactionEnd()`. The syntax is:

```
onTransactionEnd(function callback)
```

The parameters are:

`finction` *`callback`*
The callback function to be executed after every transaction ends.

Logging Transaction Data

To tell the `MetricsService` where to send your transaction data, register a beacon. When a transaction ends, the `MetricsService` looks for a registered beacon to send the data.

SEE ALSO:

Adding Performance Marks

Logging Data with Beacons

Adding Performance Marks

A mark measures a specific event. Use a mark to measure an interval of a larger transaction.

Starting a Mark

To start a mark, use `$A.metricsService.markStart()`. The syntax is:

```
markStart(String ns, String name, Object context)
```

The parameters are:

`String` *`ns`*
: Optional. Mark namespace. You can use any value. This parameter doesn't have anything to do with a component's namespace though you can use the component's namespace as a high-level identifier.

`String` *`name`*
: Mark name. You can use any value, such as a component or action's name.

`Object` *`context`*
: Optional custom data to log.

To add a mark that doesn't have a separate start and end time, use `$A.metricsService.mark()`.

Ending a Mark

To end a mark, use `$A.metricsService.markEnd()`. The syntax is:

```
markEnd(String ns, String name, Object context)
```

The parameters are:

`String` *`ns`*
: Optional. Mark namespace. The value must match the ***`ns`*** value in `markStart()`.

`String` *`name`*
: Mark name. The value must match the ***`ns`*** value in `markStart()`.

`Object` *`context`*
: Optional custom data to log.

SEE ALSO:

Adding Performance Transactions

Logging Data with Beacons

A beacon is a component that receives the metrics and sends them somewhere for storage. A beacon abstracts the transport layer for sending collected metrics and transactions.

A beacon must contain a `sendData()` function that encapsulates all data logging. The beacon markup uses an `<aura:method>` tag with `id` and `transaction` attributes to define the `sendData()` function. For example:

```
<aura:method name="sendData">
    <aura:attribute name="id" type="Object" />
    <aura:attribute name="transaction" type="Object" />
</aura:method>
```

The `sendData()` function can contain any custom logic to log the performance data. Typically, it calls a server-side caboose action to log the data.

To register a beacon for all transactions, add this JavaScript code:

```
$A.metricsService.registerBeacon(component);
```

The `init` handler for a component is a typical place to register a beacon.

SEE ALSO:

- Adding Performance Transactions
- aura:method
- Calling a Server-Side Action
- Caboose Actions
- Invoking Actions on Component Initialization

Abstracting Measurement with Plugins

A plugin hooks into the code being measured and enables you to instrument your functional code without adding performance marks directly in your functional code. Add the performance marks in the plugin so that your functional code doesn't get littered with marks. This leads to a clean separation between functional code and instrumentation code that measures the performance of the functional code.

A plugin uses AOP (aspect-oriented programming) and the `MetricsService` API calls to hook into the code being measured.

Create a plugin when you want to test the performance of your code without adding marks in the functional code. The framework has several plugins for performance testing of different features. The plugins can be disabled in `PROD` mode so that the instrumentation doesn't adversely affect performance.

Tip: Your plugin code runs on every call to an instrumented function. Be selective in using plugins in `PROD` mode to limit the instrumentation to the metrics you care about.

You don't have to create your own plugins unless you want to instrument a complex code path. Alternatively, consider adding a plugin if you don't have write access to the underlying code that you want to measure, or if the code is called from multiple places and you don't want to add marks in all those places.

These are the most important methods that you can customize for your plugin.

`initialize`
Called by `MetricsService` before bootstrapping the framework so you can bind your before and after hooks using the `instrument()` method of `MetricsService`.

`enable`
Enables the plugin.

`disable`
Disables the plugin.

`postProcess`
The method called before sending the transaction in `DEV` mode. Add logic to massage the payload that the transaction aggregates.

Tip: The best way to understand plugins is to look at some existing code. For an example of a plugin, see `ClientServiceMetricsPlugin.js` in the open source git repo.

The plugin uses `$A.metricsService.registerPlugin()` to register itself.

```
// Register the plugin
$A.metricsService.registerPlugin({
    "name"   : ClientServiceMetricsPlugin.NAME,
    "plugin" : ClientServiceMetricsPlugin
});
```

You can add a plugin in any file as long as it calls `$A.metricsService.registerPlugin()`.

SEE ALSO:

Adding Performance Marks

End-to-End `MetricsService` Example

Let's tie it all together by creating a beacon and a sample component that creates a transaction and a mark. These metrics are sent to the beacon.

IN THIS SECTION:

Step 1: Create a Beacon Component
Add a beacon component that receives the metrics data.

Step 2: Add a Transaction and Mark
Add a component that contains a transaction and a mark.

Step 1: Create a Beacon Component

Add a beacon component that receives the metrics data.

1. Add the markup for the beacon.

```
<!--c:metricsBeacon-->
<aura:component>
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <aura:method name="sendData">
        <aura:attribute name="id" type="Object"/>
        <aura:attribute name="transaction" type="Object"/>
    </aura:method>
</aura:component>
```

The component doesn't have any UI output. It just sets up the `sendData` method for the beacon.

2. Add the client-side controller code.

```
/*metricsBeaconController.js*/
({
    doInit : function(component, event, helper) {
        $A.metricsService.registerBeacon(component);
    },

    sendData: function (cmp, event, helper) {
        var args = event.getParams().arguments;

        // Log to console as an example.
        // In production-quality code, data would be logged
        // and persisted with a caboose server-side action
        console.log("in beacon: ", args);
    }
})
```

The `doInit()` function registers the beacon. The `init` event configured in the beacon's markup triggers `doInit()`.

This simple beacon logs the output but in production-quality code, you could persist the performance data for analysis.

SEE ALSO:

Logging Data with Beacons

Step 2: Add a Transaction and Mark

Add a component that contains a transaction and a mark.

1. Add the component markup.

```
<!--c:metricsSample-->
<aura:component>
    <c:metricsBeacon/>
    <ui:button label="Log Transaction" press="{!c.logTrans}"/>
</aura:component>
```

The markup includes a reference to the beacon component so that the beacon is loaded.

When you click the button, it will log a transaction after we set up the client-side controller.

2. Add the client-side controller code.

```
/*metricsSampleController.js*/
({
    logTrans : function(cmp) {
        console.log("in logTrans");

        $A.metricsService.transactionStart('c', 'sampleTrans',
          {context : Date.now()}
        );
        // imagine a server call here

        $A.metricsService.markStart('c', 'sampleMark');
        // imagine some client-side component creation here
        $A.metricsService.markEnd('c', 'sampleMark');

        $A.metricsService.transactionEnd('c', 'sampleTrans');
    }
})
```

The client-side controller creates a transaction and mark. This sample demonstrates the scaffolding and doesn't include production-quality code that would make a server call and perform some client-side processing. Adding a mark within the transaction can give you more insight into where time is consumed in a transaction.

3. Click the **Log Transaction** button in `c:metricsSample`.
   Look in the browser's console log for the metrics.

SEE ALSO:

Adding Performance Transactions

Adding Performance Marks

CHAPTER 11 Plugging in Custom Code with Adapters

In this chapter ...

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.
`ExceptionAdapter`	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 `$Label`, 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 12 Accessing Components from Non-Aura Containers

In this chapter ...

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 enables you to plug Aura components into HTML containers.

1. Create a Java instance of the Aura Integration Service.

```
IntegrationService svc = Aura.getIntegrationService();
```

2. Create an integration, which enables 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 enables 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"`.
   - 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.
   - For the sixth argument, pass a boolean set to `true` to use asynchronous component creation for the injected component instead of the default method of printing the component HTML to the page. The asynchronous option is more performant if you are injecting multiple components.

```
Map<String, Object> attributes = Maps.newHashMap();
attributes.put("label", "Click Me");
Appendable out = new StringBuffer();
boolean async = true;
integration.injectComponent("ui:button", attributes, "button1", "div1", out, async);
```

Example: This 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();
boolean async = true;
integration.injectComponent("ui:button", attributes, "button1", "div1", out, async);
```

CHAPTER 13 Customizing Data Type Conversions

In this chapter ...

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 *`Type1`*`To`*`Type2`*`Converter`. 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 JsonSerializable {
    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();
    }
}
```

CHAPTER 14 Reference

In this chapter ...

This section contains reference documentation including details of the various tags available in the framework.

Reference Doc App

The Reference tab of the doc app includes more reference information, including descriptions and source for the out-of-the-box components that come with the framework, as well as the JavaScript API.

Supported aura:attribute Types

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

Attribute Name	Type	Description
`access`	String	Indicates whether the attribute can be used outside of its own namespace. Possible values are `internal` (default), `private`, `public`, and `global`.
`name`	String	Required. The name of the attribute. For example, if you set `<aura:attribute name="isTrue" type="Boolean" />` on a component called `aura:newCmp`, you can set this attribute when you instantiate the component; for example,`<aura:newCmp isTrue="false" />`.
`type`	String	Required. The type of the attribute. For a list of basic types supported, see Basic Types.
`default`	String	The default value for the attribute, which can be overwritten as needed. When setting a default value, expressions using the `$Label`, `$Locale`, and `$Browser` global value providers are supported. Alternatively, to set a dynamic default, use an `init` event. See Invoking Actions on Component Initialization on page 155.
`required`	Boolean	Determines if the attribute is required. The default is `false`.
`description`	String	A summary of the attribute and its usage.
`serializeTo`	String	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 `SERVER`, `BOTH`, or `NONE`. The default is `BOTH`. Specify `SERVER` if you don't want to serialize the attribute to the client. Specify `NONE` if you don't need the attribute to be serialized at all. For example, use `NONE` 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 `<aura:attribute name="myObj" type="List" serializeTo="NONE">`.

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

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

Note: Although type values are case insensitive, case sensitivity should be respected as your markup interacts with JavaScript, CSS, and 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 the framework is written in Java, defaults, such as maximum size for a number, for these basic types are defined by the Java objects that they map to.

type	Example	Description
`Boolean`	`<aura:attribute name="showDetail" type="Boolean" />`	Valid values are `true` or `false`. To set a default value of `true`, add `default="true"`.
`Date`	`<aura:attribute name="startDate" type="Date" />`	A date corresponding to a calendar day in the format yyyy-mm-dd. The hh:mm:ss portion of the date is not stored. To include time fields, use `DateTime` instead.
`DateTime`	`<aura:attribute name="lastModifiedDate" type="DateTime" />`	A date corresponding to a timestamp. It includes date and time details with millisecond precision.
`Decimal`	`<aura:attribute name="totalPrice" type="Decimal" />`	`Decimal` values can contain fractional portions (digits to the right of the decimal). Maps to java.math.BigDecimal. `Decimal` is better than `Double` for maintaining precision for floating-point calculations. It's preferable for currency fields.
`Double`	`<aura:attribute name="widthInchesFractional" type="Double" />`	`Double` values can contain fractional portions. Maps to java.lang.Double. Use `Decimal` for currency fields instead.
`Integer`	`<aura:attribute name="numRecords" type="Integer" />`	`Integer` values can contain numbers with no fractional portion. Maps to java.lang.Integer, which defines its limits, such as maximum size.
`Long`	`<aura:attribute name="numSwissBankAccount" type="Long" />`	`Long` values can contain numbers with no fractional portion. Maps to java.lang.Long, which defines its limits, such as maximum size. Use this data type when you need a range of values wider than those provided by `Integer`.
`String`	`<aura:attribute name="message" type="String" />`	A sequence of characters.

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

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

To retrieve a string array from your Java controller, use `List<String>`.

```
public List<String> getStringList() {
    List<String> colors = new List<>();
    colors.add("red");
    colors.add("blue");
    return colors;
}
```

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();
```

Checking for Types

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

Collection Types

Here are the supported collection type values.

type	Example	Description
type[] (Array)	`<aura:attribute name="colorPalette" type="String[]" default="['red', 'green', 'blue']" />`	An array of items of a defined type.

type	Example	Description
`List`	`<aura:attribute name="colorPalette" type="List" default="['red', 'green', 'blue']" />`	An ordered collection of items.
`Map`	`<aura:attribute name="sectionLabels" type="Map" default="{ a: 'label1', b: 'label2' }" />`	A collection that maps keys to values. A map can't contain duplicate keys. Each key can map to at most one value. Defaults to an empty object, `{}`. Retrieve values by using `cmp.get("v.sectionLabels")['a']`.
`Set`	`<aura:attribute name="collection" type="Set" default="['red', 'green', 'blue']" />`	A collection that contains no duplicate elements. The order for set items is not guaranteed. For example, `"red,green,blue"` might be returned as `"blue,green,red"`.

Checking for Types

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

Setting List Items

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

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

```
<aura:attribute name="numbers" type="List"/>
<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, 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.

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" />
```

Framework-Specific Types

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

type	Example	Description
`Aura.Component`	N/A	A single component. We recommend using `Aura.Component[]` instead.

`Aura.Component[]`	`<aura:attribute name="detail" type="Aura.Component[]"/>`
	To set a default value for `type="Aura.Component[]"`, put the default markup in the body of `aura:attribute`. For example:
	`<aura:component>`
	`    <aura:attribute name="detail" type="Aura.Component[]">`
	`    <p>default paragraph1</p>`
	`    </aura:attribute>`
	`    Default value is: {!v.detail}`
	`</aura:component>`
		Use this type to set blocks of markup. An attribute of type `Aura.Component[]` is called a facet.

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

SEE ALSO:

Component Body

Component Facets

Using the Action Type

An `Aura.Action` is a reference to an action in the framework. 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.

This sample uses `Aura.Action`.

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"`.

subListRow.cmp

```
<aura:component extends="docsample:listRow">
    ...
    <aura:set attribute="onclick" value="{!c.openRecord}"/>
    ...
</aura:component>
```

The `subListRow` component extends the `listRow` component and sets the value for the `onclick` attribute in `listRow` to `{!c.openRecord}`, which is a reference to an action in the client-side controller for `subListRow.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

aura:application

An app is a special top-level component whose markup is in a `.app` file.

The markup looks similar to HTML and can contain components as well as a set of supported HTML tags. The `.app` file is a standalone entry point for the app and enables you to define the overall application layout, style sheets, and global JavaScript includes. It starts with the top-level `<aura:application>` tag, which contains optional system attributes. These system attributes tell the framework how to configure the app.

System Attribute	Type	Description
`access`	String	Indicates whether the app can be extended by another app outside of a namespace. Possible values are `internal` (default), `public`, and `global`.
`controller`	String	The server-side controller class for the app. The format is `java://<package.class>`.
`description`	String	A brief description of the app.
`extends`	Component	The app to be extended, if applicable. For example, `extends="namespace:yourApp"`.
`extensible`	Boolean	Indicates whether the app is extensible by another app. Defaults to `false`.
`implements`	String	A comma-separated list of interfaces that the app implements.

System Attribute	Type	Description
`locationChangeEvent`	Event	The framework monitors the location of the current window for changes. If the `#` value in a URL changes, the framework fires an application event. The `locationChangeEvent` defines this event. The default value is `aura:locationChange`. The `locationChange` event has a single attribute called `token`, which is set with everything after the `#` value in the URL.
`model`	String	The model class used to initialize data for the app. The format is `java://<package.class>`.
`preload`	String	Deprecated. Use the aura:dependency tag instead. If you use the `preload` system attribute, the framework internally converts the value to `<aura:dependency>` tags.
`render`	String	Renders the component using client-side or server-side renderers. If not provided, the framework determines any dependencies and whether the application should be rendered client-side or server-side. Valid options are `client` or `server`. The default is `auto`. For example, specify `render="client"` if you want to inspect the application on the client-side during testing.
`renderer`	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, the framework 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 **`<appName>`**`Renderer.js`, if it exists in your application bundle.
`template`	Component	The name of the template used to bootstrap the loading of the framework and the app. The default value is `aura:template`. You can customize the template by creating your own component that extends the default template. For example: `<aura:component extends="aura:template" ... >`
`tokens`	String	A comma-separated list of tokens bundles for the application. For example, `tokens="ns:myAppTokens"`. Tokens make it easy to ensure that your design is consistent, and even easier to update it as your design evolves. Define the token values once and reuse them throughout your application.
`useAppcache`	Boolean	Specifies whether to use the application cache. Valid options are `true` or `false`. Defaults to `false`.

`aura:application` also includes a `body` attribute defined in a `<aura:attribute>` tag. Attributes usually control the output or behavior of a component, but not the configuration information in system attributes.

Attribute	Type	Description
`body`	`Component[]`	The body of the app. In markup, this is everything in the body of the tag.

SEE ALSO:

- URL-Centric Navigation
- Creating Apps
- Using the AppCache
- Application Access Control

aura:component

The root of the component hierarchy. Provides a default rendering implementation.

Components are the functional units of Aura, which encapsulate modular and reusable sections of UI. They can contain other components or HTML markup. The public parts of a component are its attributes and events. Aura provides out-of-the-box components in the `aura` and `ui` namespaces.

Every component is part of a namespace. For example, the `button` component is saved as `button.cmp` in the `ui` namespace can be referenced in another component with the syntax `<ui:button label="Submit"/>`, where `label="Submit"` is an attribute setting.

To create a component, follow this syntax.

```
<aura:component>
    <!-- Optional coponent attributes here -->
    <!-- Optional HTML markup -->
    <div class="container">
        Hello world!
        <!-- Other components -->
    </div>
</aura:component>
```

A component has the following optional attributes.

Attribute	Type	Description
`abstract`	Boolean	Set to `true` if the component is abstract. The default is `false`.
`access`	String	Indicates whether the component can be used outside of its own namespace. Possible values are `internal` (default), `public`, and `global`.
`aura:flavorable`	Boolean	Set to `true` if the component is flavorable. The default is `false`.
`controller`	String	The server-side controller class for the component. The format is `java://<package.class>`.
`defaultFlavor`	String	The comma-separated list of flavor names in the component bundle, defined in the `<componentName>Flavors.css` file.

Attribute	Type	Description
`description`	String	A description of the component.
`extends`	Component	The component to be extended.
`extensible`	Boolean	Set to `true` if the component can be extended. The default is `false`.
`helper`	String	The external JavaScript helper file to use. If you use an external helper file, the helper methods in your component bundle will not be accessible. The format is `js://namespace.component`.
`implements`	String	A comma-separated list of interfaces that the component implements.
`isTemplate`	Boolean	Set to `true` if the component is a template. The default is `false`. A template must have `isTemplate="true"` set in its `<aura:component>` tag. `<aura:component isTemplate="true" extends="aura:template">`
`model`	String	The model class used to initialize data for the component. The format is `java://<package.class>`.
`provider`	String	The JavaScript or Java provider, in the format `java://<package.class>` or `js://namespace.component`. A provider enables you to use an abstract component in markup. Defining a server-side provider overrides any client-side provider in the component bundle.
`render`	String	Renders the component using client-side or server-side renderers. If not provided, the framework determines any dependencies and whether the component should be rendered client- or server-side. Valid options are `client` or `server`. The default is `auto`. Specify this attribute in the top-level component. For example, specify `render="client"` if you want to inspect the component on the client-side during testing.
`support`	String	The support level for the component. Valid options are `PROTO`, `DEPRECATED`, `BETA`, or `GA`.
`template`	Component	The template for this component. A template bootstraps loading of the framework and app. The default template is `aura:template`. You can customize the template by creating your own component that extends the default template. For example: `<aura:component extends="aura:template" ... >`
`whitespace`	String	Preserves or removes unnecessary whitespace in the component markup. Valid options are `preserve` or `optimize`. The default is `optimize`.

aura:component 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.

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. Create a dynamic library resolver that points to the client library.

The <aura:clientLibrary> tag includes these system attributes.

System Attribute	Description
modes	A comma-separated list of modes that use the client library. If no value is set, the library is available for all modes.
name	The name of a ClientLibraryResolver that provides the URL. The name attribute is useful if the location or URL of the library needs to be dynamically generated. The name attribute is required if the url attribute is not specified; otherwise, it's ignored. See Add a Client Library Resolver on page 269.
type	The type of library. Values are CSS, or JS for JavaScript.

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 an external JavaScript library with a name of MadLib:

```
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 External JavaScript Libraries

aura:dependency

The `<aura:dependency>` tag enables you to declare dependencies that can't easily be discovered by the framework.

The 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`	The resource that the component depends on. For example, `resource="markup://sampleNamespace:sampleComponent"` refers to the `sampleComponent` in the `sampleNamespace` namespace. Use an asterisk (`*`) in the resource name for wildcard matching. For example, `resource="markup://sampleNamespace:*"` matches everything in the namespace; `resource="markup://sampleNamespace:input*"` matches everything in the namespace that starts with `input`. Don't use an asterisk (`*`) in the namespace portion of the resource name. For example, `resource="markup://sample*:sampleComponent"` is not supported.

System Attribute	Description
`type`	The type of resource that the component depends on. The default value is `COMPONENT`. Use `type="*"` to match all types of resources. The most commonly used values are: • `COMPONENT` • `APPLICATION` • `EVENT` Use a comma-separated list for multiple types; for example: `COMPONENT,APPLICATION`.

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
`access`	String	Indicates whether the event can be extended or used outside of its own namespace. Possible values are `internal` (default), `public`, and `global`.
`description`	String	A description of the event.
`extends`	Component	The event to be extended. For example, `extends="namespace:myEvent"`.
`type`	String	Required. Possible values are `COMPONENT` or `APPLICATION`.
`support`	String	The support level for the event. Valid options are `PROTO`, `DEPRECATED`, `BETA`, or `GA`.

SEE ALSO:

Communicating with Events

Event Access Control

aura:if

`aura:if` renders the content within the tag if the `isTrue` attribute evaluates to true.

The framework evaluates the `isTrue` expression 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.

Attribute Name	Type	Description
`else`	ComponentDefRef[]	The markup to render when `isTrue` evaluates to false. Set this attribute using the `aura:set` tag.
`isTrue`	string	Required. An expression that determines whether the content is displayed. If it evaluates to `true`, the content is displayed.

Example

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

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

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

SEE ALSO:

Best Practices for Conditional Markup

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 its own 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.

Attribute	Type	Description
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.

Attribute Name	Type	Description
body	ComponentDefRef[]	Required. Template to use when creating components for each iteration. You can put any markup in the body. A ComponentDefRef[] stores the metadata of the component instances to create on each iteration, and each instance is then stored in realbody.
end	Integer	The index of the collection to stop at (exclusive).
forceServer	Boolean	Force a server request for the component body. Set to true if the iteration requires any server-side creation. The default is false.
indexVar	String	The variable name to use for the index of each item inside the iteration.
items	List	Required. The collection of data to iterate over.
realbody	Component[]	Do not use. Any value set is ignored. Placeholder for body rendering.
start	Integer	The index of the collection to start at (inclusive).
var	String	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 with an HTML meter tag.

```
<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.

SEE ALSO:

Client-Side Runtime Binding of Components

Server-Side Runtime Binding of Components

aura:method

Use `<aura:method>` to define a method as part of a component's API. This enables you to directly call a method in a component's client-side controller instead of firing and handling a component event. Using `<aura:method>` simplifies the code needed for a parent component to call a method on a child component that it contains.

The `<aura:method>` tag has these system attributes.

Attribute	Type	Description
`name`	`String`	The method name. Use the method name to call the method in JavaScript code. For example: `cmp.sampleMethod(param1);`
`action`	`Expression`	The client-side controller action to execute. For example: `action="{!c.sampleAction}"` `sampleAction` is an action in the client-side controller. If you don't specify an `action` value, the controller action defaults to the value of the method `name`.
`access`	`String`	The access control for the method. Valid values are: • **internal**—Any component in a system namespace can call the method. A system namespace is a privileged namespace that has access to all components. This is the default access level. • **public**—Any component in the same namespace can call the method. • **global**—Any component in any namespace can call the method.
`description`	`String`	The method description.

Declaring Parameters

An `<aura:method>` can optionally include parameters. Use an `<aura:attribute>` tag within an `<aura:method>` to declare a parameter for the method. For example:

```
<aura:method name="sampleMethod" action="{!c.doAction}"
  description="Sample method with parameters">
    <aura:attribute name="param1" type="String" default="parameter 1"/>
    <aura:attribute name="param2" type="Object" />
</aura:method>
```

Note: You don't need an `access` system attribute in the `<aura:attribute>` tag for a parameter.

Creating a Handler Action

This handler action shows how to access the arguments passed to the method.

```
({
    doAction : function(cmp, event) {
        var params = event.getParam('arguments');
        if (params) {
            var param1 = params.param1;
            // add your code here
        }
    }
})
```

Retrieve the arguments using `event.getParam('arguments')`. It returns an object if there are arguments or an empty array if there are no arguments.

SEE ALSO:

Calling Component Methods

Component Events

aura:renderIf

Deprecated. Use `aura:if` instead.

`aura:renderIf` renders the content within the tag if the `isTrue` attribute evaluates to `true`. The previous advice was to 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. This advice is no longer relevant. Always use `aura:if` instead.

Attribute Name	Type	Description
`else`	Component[]	The markup to render when `isTrue` evaluates to `false`. Set this attribute using the `aura:set` tag.
`isTrue`	String	Required. An expression that determines whether the content is displayed. If it evaluates to `true`, the content is displayed.

Example

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

```
<aura:attribute name="edit" type="Boolean" default="true">
<aura:renderIf isTrue="{!v.edit}">
```

```
    <ui:button label="Edit"/>
    <aura:set attribute="else">
        You can’t edit this.
        <!-- Imagine some components here that need to be created on the server -->
    </aura:set>
</aura:renderIf>
```

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

We recommend using `aura:if` instead if the `else` attribute is rarely displayed or if it doesn’t include components that need to be created on the server.

SEE ALSO:

Best Practices for Conditional Markup

aura:if

aura:set

Use `<aura:set>` in markup to set the value of an attribute inherited from a super component, event, or interface.

To learn more, see:

- Setting Attributes Inherited from a Super Component
- Setting Attributes on a Component Reference
- Setting Attributes Inherited from an Interface

Setting Attributes Inherited from a Super Component

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

Let's look at an example. Here is the `c:setTagSuper` component.

```
<!--c:setTagSuper-->
<aura:component extensible="true">
    <aura:attribute name="address1" type="String" />
    setTagSuper address1: {!v.address1}<br/>
</aura:component>
```

`c:setTagSuper` outputs:

```
setTagSuper address1:
```

The `address1` attribute doesn't output any value yet as it hasn't been set.

Here is the `c:setTagSub` component that extends `c:setTagSuper`.

```
<!--c:setTagSub-->
<aura:component extends="c:setTagSuper">
    <aura:set attribute="address1" value="808 State St" />
</aura:component>
```

`c:setTagSub` outputs:

```
setTagSuper address1: 808 State St
```

`sampleSetTagExc:setTagSub` sets a value for the `address1` attribute inherited from the super component, `c:setTagSuper`.

Warning: This usage of `<aura:set>` works for components and abstract components, but it doesn't work for interfaces. For more information, see Setting Attributes Inherited from an Interface on page 278.

If you're using a component by making a reference to it in your component, you can set the attribute value directly in the markup. For example, `c:setTagSuperRef` makes a reference to `c:setTagSuper` and sets the `address1` attribute directly without using `aura:set`.

```
<!--c:setTagSuperRef-->
<aura:component>
    <c:setTagSuper address1="1 Sesame St" />
</aura:component>
```

`c:setTagSuperRef` outputs:

```
setTagSuper address1: 1 Sesame St
```

SEE ALSO:

Component Body

Inherited Component Attributes

Setting Attributes on a Component Reference

Setting Attributes on a Component Reference

When you include another component, such as `<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="Save">
    <aura:set attribute="buttonTitle" value="Click to save the record"/>
</ui:button>
```

This is equivalent to:

```
<ui:button label="Save" buttonTitle="Click to save the record" />
```

The latter syntax without `aura:set` makes more sense in this simple example. You can also use this simpler syntax in component references to set values for attributes that are inherited from parent components.

`aura:set` is more useful when you want to set markup as the attribute value. For example, this sample specifies the markup for the `else` attribute in the `aura:if` tag.

```
<aura:component>
    <aura:attribute name="display" type="Boolean" default="true"/>
    <aura:if isTrue="{!v.display}">
        Show this if condition is true
        <aura:set attribute="else">
           <ui:button label="Save" press="{!c.saveRecord}" />
        </aura:set>
    </aura:if>
</aura:component>
```

SEE ALSO:

Setting Attributes Inherited from a Super Component

Setting Attributes Inherited from an Interface

To set the value of an attribute inherited from an interface, redefine the attribute in the component and set its default value. Let's look at an example with the `c:myIntf` interface.

```
<!--c:myIntf-->
<aura:interface>
    <aura:attribute name="myBoolean" type="Boolean" default="true" />
</aura:interface>
```

This component implements the interface and sets `myBoolean` to `false`.

```
<!--c:myIntfImpl-->
<aura:component implements="c:myIntf">
    <aura:attribute name="myBoolean" type="Boolean" default="false" />

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

System Event Reference

System events are fired by the framework during its lifecycle. You can handle these events in your Lightning apps or components, and within Salesforce1. For example, these events enable you to handle attribute value changes, URL changes, or when the app or component is waiting for a server response.

aura:doneRendering

Indicates that the initial rendering of the root application or root component has completed.

This event is automatically fired if no more components need to be rendered or rerendered due to any attribute value changes. The `aura:doneRendering` event is handled by a client-side controller. A component can have only one `<aura:handler event="doneRendering">` tag to handle this event.

```
<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:component>
    <aura:handler event="aura:doneRendering" action="{!c.doneRendering}"/>
    <aura:attribute name="isDoneRendering" type="Boolean" default="false"/>
    <!-- Other component markup here -->
    <p>My component</p>
</aura:component>
```

This client-side controller checks that the `aura:doneRendering` event has been fired only once.

```
({
  doneRendering: function(cmp, event, helper) {
    if(!cmp.get("v.isDoneRendering")){
      cmp.set("v.isDoneRendering", true);
      //do something after component is first rendered
    }
  }
})
```

Note: When `aura:doneRendering` is fired, `component.isRendered()` returns `true`. To check if your element is visible in the DOM, use utilities such as `component.getElement()`, `component.hasClass()`, or `element.style.display`.

The `aura:doneRendering` handler contains these required attributes.

Attribute Name	Type	Description
`event`	String	The name of the event, which must be set to `aura:doneRendering`.
`action`	Object	The client-side controller action that handles the event.

aura:doneWaiting

Indicates that the app or component is done waiting for a response to a server request. This event is preceded by an `aura:waiting` event. This event is fired after `aura:waiting`.

This event is automatically fired if no more response from the server is expected. The `aura:doneWaiting` event is handled by a client-side controller. A component can have only one `<aura:handler event="aura:doneWaiting">` tag to handle this event.

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

This example hides a spinner when `aura:doneWaiting` is fired.

```
<aura:component>
     <aura:handler event="aura:doneWaiting" action="{!c.hideSpinner}"/>
    <!-- Other component markup here -->
    <center><ui:spinner aura:id="spinner"/></center>
</aura:component>
```

This client-side controller fires an event that hides the spinner.

```
({
    hideSpinner : function (component, event, helper) {
        var spinner = component.find('spinner');
        var evt = spinner.get("e.toggle");
        evt.setParams({ isVisible : false });
        evt.fire();
    }
})
```

The `aura:doneWaiting` handler contains these required attributes.

Attribute Name	Type	Description
`event`	String	The name of the event, which must be set to `aura:doneWaiting`.
`action`	Object	The client-side controller action that handles the event.

aura:locationChange

Indicates that the hash part of the URL has changed.

This event is automatically fired when the hash part of the URL has changed, such as when a new location token is appended to the hash. The `aura:locationChange` event is handled by a client-side controller. A component can have only one `<aura:handler event="aura:locationChange">` tag to handle this event.

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

This client-side controller handles the `aura:locationChange` event.

```
({
    update : function (component, event, helper) {
        // Get the new location token from the event
        var loc = event.getParam("token");
        // Do something else
    }
})
```

The `aura:locationChange` handler contains these required attributes.

Attribute Name	Type	Description
`event`	String	The name of the event, which must be set to `aura:locationChange`.
`action`	Object	The client-side controller action that handles the event.

The `aura:locationChange` event contains these attributes.

Attribute Name	Type	Description
`token`	String	The hash part of the URL.
`querystring`	Object	The query string portion of the hash.

aura:systemError

Indicates that an error has occurred.

This event is fired when a `$A.auraFriendlyError` error is thrown. Handle the `aura:systemError` event in a client-side controller. A component can have only one `<aura:handler event="aura:systemError">` tag in markup to handle this event.

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

Throw the error using `$A.auraFriendlyError("error message here")`.

Set the error message in the `message` property of `AuraFriendlyError`, which corresponds to the first argument of `$A.auraFriendlyError()`. Set an optional object with more context in the `data` property of `AuraFriendlyError`.

```
({
    throwError : function(cmp, event){
        // error is an instance of AuraFriendlyError
        // argument sets the message property of AuraFriendlyError
        var error = new $A.auraFriendlyError("This is a sample error.");
        // set an optional error data object
        error.data = {
            "moreErrorData1": "more1",
            "moreErrorData2": "more2",
        };
        throw error;
    },
})
```

Set `event["handled"]=true` in the client-side controller action that handles the `aura:systemError` event to indicate that you're providing your own error handler.

The `aura:handler` tag for the `aura:systemError` event contains these required attributes.

Attribute Name	Type	Description
`event`	String	The name of the event, which must be set to `aura:systemError`.
`action`	Object	The client-side controller action that handles the event.

The `aura:systemError` event contains these attributes. You can retrieve the attribute values using `event.getParam("`***`attributeName`***`")`.

Attribute Name	Type	Description
`message`	String	The error message.
`error`	String	The name of the error, `AuraFriendlyError`.
`auraError`	Object	The `AuraFriendlyError` error object.

SEE ALSO:

Throwing and Handling Errors

aura:valueChange

Indicates that an attribute value has changed.

This event is automatically fired when an attribute value changes. The `aura:valueChange` event is handled by a client-side controller. A component can have multiple `<aura:handler name="change">` tags to detect changes to different attributes.

```
<aura:handler name="change" value="{!v.items}" action="{!c.itemsChange}"/>
```

This example updates a Boolean value, which automatically fires the `aura:valueChange` event.

```
<aura:component>
    <aura:attribute name="myBool" type="Boolean" default="true"/>

    <!-- Handles the aura:valueChange event -->
    <aura:handler name="change" value="{!v.myBool}" action="{!c.handleValueChange}"/>
    <ui:button label="change value" press="{!c.changeValue}"/>
</aura:component>
```

These client-side controller actions trigger the value change and handle it.

```
({
    changeValue : function (component, event, helper) {
      component.set("v.myBool", false);
    },

    handleValueChange : function (component, event, helper) {
        // handle value change
        console.log("old value: " + event.getParam("oldValue"));
        console.log("current value: " + event.getParam("value"));
    }
})
```

The `valueChange` event gives you access to the previous value (`oldValue`) and the current value (`value`) in the handler action. In this example, `oldValue` returns `true` and `value` returns `false`.

The `change` handler contains these required attributes.

Attribute Name	Type	Description
`name`	String	The name of the handler, which must be set to `change`.
`value`	Object	The attribute for which you want to detect changes.
`action`	Object	The client-side controller action that handles the value change.

SEE ALSO:

Detecting Data Changes with Change Handlers

aura:valueDestroy

Indicates that a component has been destroyed.

This event is automatically fired when a component is being destroyed. The `aura:valueDestroy` event is handled by a client-side controller. A component can have only one `<aura:handler name="destroy">` tag to handle this event.

```
<aura:handler name="destroy" value="{!this}" action="{!c.handleDestroy}"/>
```

This client-side controller handles the `aura:valueDestroy` event.

```
({
    valueDestroy : function (component, event, helper) {
      var val = event.getParam("value");
      // Do something else here
    }
})
```

Let's say that you are viewing a component in Salesforce1. The `aura:valueDestroy` event is triggered when you tap on a different menu item on the Salesforce1 navigation menu, and your component is destroyed. In this example, the `value` parameter in the event returns the component that's being destroyed.

The `<aura:handler>` tag for the `aura:valueDestroy` event contains these required attributes.

Attribute Name	Type	Description
`name`	String	The name of the handler, which must be set to `destroy`.
`value`	Object	The value for which you want to detect the event for. The value that is being destroyed. Always set `value="{!this}"`.
`action`	Object	The client-side controller action that handles the destroy event.

The `aura:valueDestroy` event contains these attributes.

Attribute Name	Type	Description
`value`	String	The component being destroyed, which is retrieved via `event.getParam("value")`.

aura:valueInit

Indicates that an app or component has been initialized.

This event is automatically fired when an app or component is initialized, prior to rendering. The `aura:valueInit` event is handled by a client-side controller. A component can have only one `<aura:handler name="init">` tag to handle this event.

```
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
```

For an example, see Invoking Actions on Component Initialization on page 155.

Note: Setting `value="{!this}"` marks this as a value event. You should always use this setting for an `init` event.

The `init` handler contains these required attributes.

Attribute Name	Type	Description
`name`	String	The name of the handler, which must be set to `init`.
`value`	Object	The value that is initialized, which must be set to `{!this}`.
`action`	Object	The client-side controller action that handles the value change.

aura:waiting

Indicates that the app or component is waiting for a response to a server request. This event is fired before `aura:doneWaiting`.

This event is automatically fired when a server-side action is added using `$A.enqueueAction()` and subsequently run, or when it's expecting a response from an Apex controller. The `aura:waiting` event is handled by a client-side controller. A component can have only one `<aura:handler event="aura:waiting">` tag to handle this event.

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

This example shows a spinner when `aura:waiting` is fired.

```
<aura:component>
     <aura:handler event="aura:waiting" action="{!c.showSpinner}"/>
    <!-- Other component markup here -->
    <center><ui:spinner aura:id="spinner"/></center>
</aura:component>
```

This client-side controller fires an event that displays the spinner.

```
({
    showSpinner : function (component, event, helper) {
        var spinner = component.find('spinner');
        var evt = spinner.get("e.toggle");
        evt.setParams({ isVisible : true });
        evt.fire();
    }
})
```

The `aura:waiting` handler contains these required attributes.

Attribute Name	Type	Description
`event`	String	The name of the event, which must be set to `aura:waiting`.
`action`	Object	The client-side controller action that handles the event.

Supported HTML Tags

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

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

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

We recommend that you use components in preference to HTML tags. For example, use `ui:button` instead of `<button>`. Components are designed with accessibility in mind so users with disabilities or those who use assistive technologies can also use your app. When you start building more complex components, the reusable out-of-the-box components can simplify your job by handling some of the plumbing that you would otherwise have to create yourself. Also, these components are secure and optimized for performance.

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

The majority of HTML5 tags are supported.

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

- `applet`
- `base`
- `basefont`

- `embed`
- `font`
- `frame`
- `frameset`
- `isindex`
- `noframes`
- `noscript`
- `object`
- `param`

Avoid `#` in the `href` Attribute of Anchor Tags

The hash mark (`#`) is a URL fragment identifier and is often used in Web development for navigation within a page. Avoid `#` in the `href` attribute of anchor tags in Aura components as it can cause unexpected navigation changes, especially in the Salesforce1 mobile app. For example, use `href=""` instead of `href="#"`.

SEE ALSO:

Supporting Accessibility

APPENDIX

CHAPTER 15 Aura Request Lifecycle

In this chapter ...

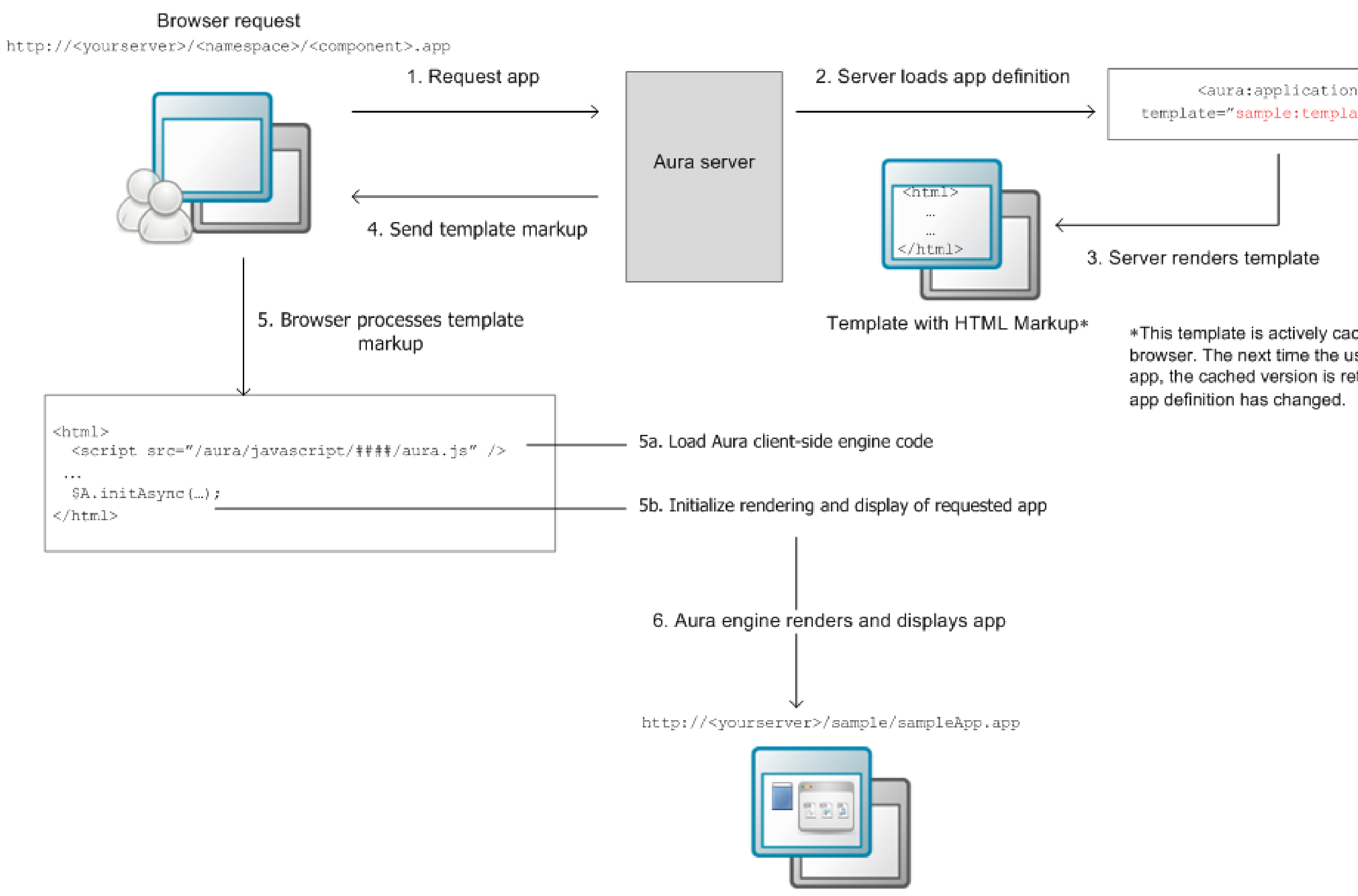
This section shows how Aura handles the initial request for an application, as well as a component request. You can use Aura without knowing these details but read on if you are curious about how things work under the covers.

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.


Browser request
http://<yourserver>/<namespace>/<component>.app
1. Request app
Aura server
2. Server loads app definition
<aura:application
template="sample:templa
<html>
...
...
</html>
3. Server renders template
4. Send template markup
Template with HTML Markup*
*This template is actively cac
browser. The next time the us
app, the cached version is ret
app definition has changed.
5. Browser processes template markup
5a. Load Aura client-side engine code
5b. Initialize rendering and display of requested app
6. Aura engine renders and displays app
http://<yourserver>/sample/sampleApp.app

```
<html>
  <script src="/aura/javascript/####/aura.js" />
 ...
  $A.initAsync(...);
</html>
```

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.

IN THIS SECTION:

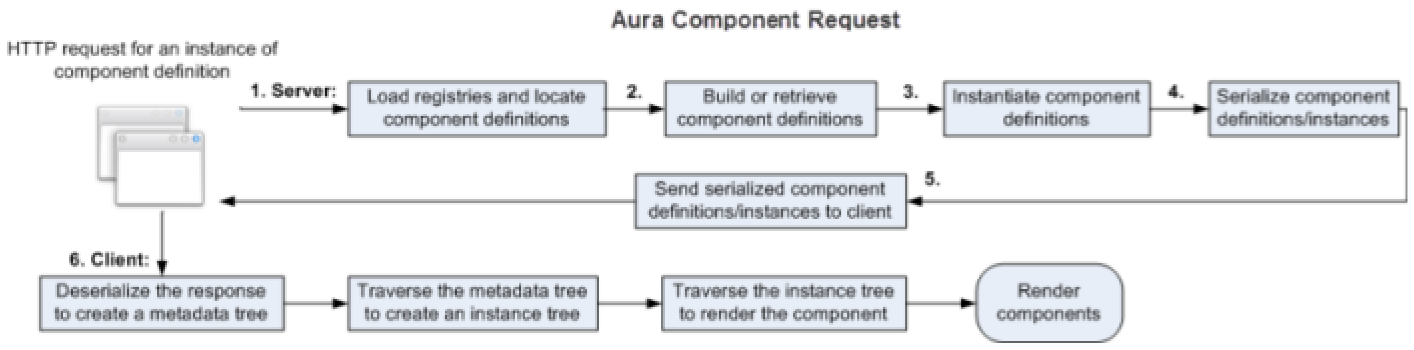
Component Request Overview

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	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. A definition's metadata can include a name, location of origin, and descriptor (DefDescriptor, the primary key of the definition).
DefDescriptor	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 `model="java://myPackage.MyClass"` 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. The qualified name for a DefDescriptor has a format of either `prefix://namespace:name` or `prefix://namespace.name`. For example, `js://ui.button`. • prefix: Defines the language, such as JavaScript or Java • namespace: Corresponds to the package name or XML namespace • name: Corresponds to the class name or local name
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.

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`.


Aura Component Request
HTTP request for an instance of component definition
1. Server:
Load registries and locate component definitions
2.
Build or retrieve component definitions
3.
Instantiate component definitions
4.
Serialize component definitions/instances
5.
Send serialized component definitions/instances to client
6. Client:
Deserialize the response to create a metadata tree
Traverse the metadata tree to create an instance tree
Traverse the instance tree to render the component
Render components

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). By default, any Map, Array, Number, Boolean, String or nulls are supported for serialization and deserialization. Other objects can provide custom serialization by implementing the `JsonSerializable` interface.

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.

As part of component initialization, client-side framework code are cached alongside your JavaScript code and CSS.

3. Traverse the instance tree to render the component.

After component initialization, 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 recursively. This adds the components to the DOM. For more information on the rendering lifecycle, see Events Fired During the Rendering Lifecycle on page 122.

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	`aura:component`	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.

Definition-related Term	Example	Description
		A definition's metadata can include a name, location of origin, and descriptor (`DefDescriptor`, the primary key of the definition). A component definition can be used by other component definitions and can extend another component definition.

Root Definition	`ComponentDef`
	`InterfaceDef`
	`EventDef`
		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	`ControllerDef`
	`ModelDef`
	`ProviderDef`
	`RendererDef`
	`StyleDef`
	`TestSuiteDef`
		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.

Subdefinition	`AttributeDef`
	`RegisterEventDef`
	`ActionDef`
	`TestCaseDef`
	`ValueDef`
		Subdefinitions can be used to define root definitions or associate definitions. They are stored directly on their parent definitions.
		For example, a `ComponentDef` can include multiple `AttributeDef` objects, and a `ControllerDef` can include multiple `ActionDef` objects.

Definition Reference	`DefRef`
	`ComponentDefRef`
	`AttributeDefRef`
		A subdefinition that points to another definition. At runtime, it can be turned into an instance of the definition to which it points.
		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.

Provider		For abstract definition types. A provider determines the concrete `ComponentDef` to instantiate for each abstract `ComponentDef`. 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

Registry-related Terms	Example	Description
		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 `model="java://myPackage.MyClass"` 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. The qualified name for a `DefDescriptor` has the format `prefix://namespace:name`. • prefix: Defines the language, such as JavaScript or Java • namespace: Corresponds to the package name or XML namespace • name: Corresponds to the class name or local name

INDEX