Lightning Components Developer's Guide

Version 32.0, Winter '15

@salesforcedocs

Last updated: November 7, 2014

CONTENTS

GETTING STARTED

CHAPTER 1 Introduction

In this chapter ...

Salesforce1 Lightning Overview

We're in an increasingly multi-device world, so we've created Lightning to make it easy to build responsive applications for any screen. Lightning includes a number of exciting tools for developers, such as:

1. Lightning components give you a client-server framework that accelerates development, as well as app performance, and is ideal for use with the Salesforce1 mobile app
2. The Lightning App Builder empowers you to build apps visually, without code, quicker than ever before using off-the-shelf and custom-built Lightning components

Using these technologies, you can seamlessly customize and easily deploy new apps to mobile devices running Salesforce1. In fact, the Salesforce1 mobile app itself was built with Lightning components.

This guide will provide you with an in-depth resource to help you create your own stand-alone Lightning apps, as well as custom Lightning components that can be used in the Salesforce1 mobile app. You will also learn how to package applications and components and distribute them in the AppExchange.

What is the Lightning Component Framework?

The Lightning Component framework 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 Apex on the server side.

Note: This release contains a beta version of the Lightning Component framework that is production quality but has some limitations.

Why Use the Lightning Component Framework?

There are many benefits of using the Lightning Component framework to build apps.

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 framework uses JSON to exchange data between the server and the client. To maximize efficiency, the server only sends data that is needed by the user.

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.

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 are responsive and provide an enjoyable user experience. The Lightning Component framework 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:

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 essentially a traditional publish-subscribe model. All components that provide a handler for the event are notified when the event is fired.

You write the handlers in JavaScript controller actions.

SEE ALSO:

Events

Handling Events with Client-Side Controllers

Browser Support

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

Browser	Notes
Google Chrome™	
Apple® Safari® 5+	For Mac OS X and iOS
Mozilla® Firefox®	

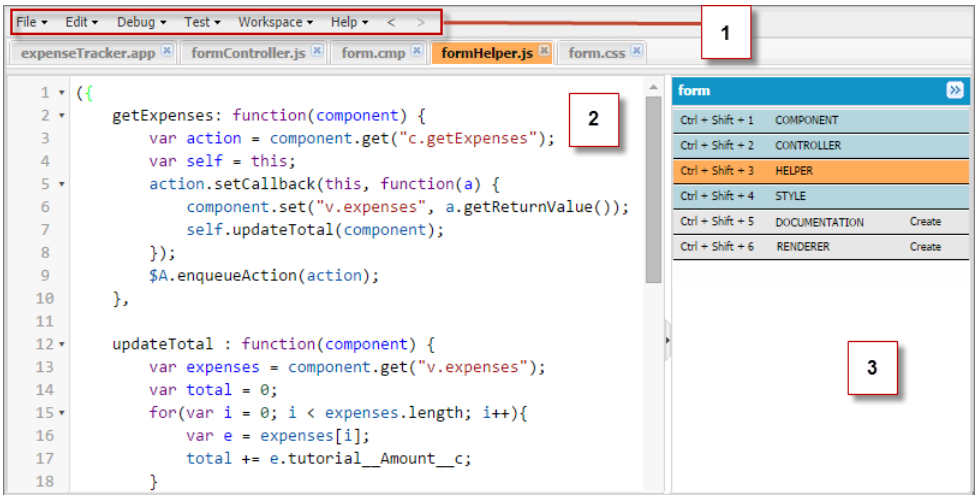
Browser	Notes
Microsoft® Internet Explorer®	We recommend using Internet Explorer 9, 10, or 11. Internet Explorer 7 and 8 may provide a degraded performance.

Note: For all browsers, you must enable JavaScript and cookies.

Using the Developer Console

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

To open the Developer Console, click ***Your name*** > **Developer Console**.



The Developer Console enables you to perform these functions.

- Use the menu bar (1) to create or open these Lightning resources.
  - Application
  - Component
  - Interface
  - Event

  Note: To create Lightning resources, you must use a Developer Edition organization that has a namespace prefix.
- Use the workspace (2) to work on your Lightning resources.
- Use the sidebar (3) to create or open client-side resources that are part of a specific component bundle.
  - Controller
  - Helper
  - Style
  - Documentation
  - Renderer

For more information on the Developer Console, see "Developer Console User Interface Overview" in the Salesforce Help.

SEE ALSO:

Component Bundles

Open Source Aura Framework

Throughout this developer guide, there are references to *Aura* components. For example, you'll see the `aura:component` tag for a component in code samples. All along we've been talking about Lightning, so what is Aura, and what's the difference? Lightning components are based on the open source Aura framework available at `https://github.com/forcedotcom/aura`. The Aura framework enables you to build apps completely independent of your data in Salesforce.

Note that the open source Aura framework has features and components that are not currently available in the Lightning Component framework. We are working to surface more of these features and components for Salesforce developers.

Online Version of this Guide

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

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

CHAPTER 2 Quick Start

In this chapter ...

The quick start steps you through building and running a simple standalone Lightning app for tracking expenses from the Developer Console.

Before You Begin

To work with standalone Lightning apps and components , make sure you have these prerequisites.

1. Create a Developer Edition organization.
2. Register a Namespace Prefix.
3. Enable Lightning Components

Create a Developer Edition Organization

The Lightning Component framework is available with a Developer Edition organization, or *DE org* for short. DE orgs are multipurpose environments with all of the features and permissions that allow you to develop, package, test, and install apps. If you don't have one, create it by following these steps.

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

Register a Namespace Prefix

Next, register a namespace prefix. Your namespace prefix must be globally unique across all Salesforce organizations. Namespace prefixes are case-insensitive and have a maximum length of 15 alphanumerical characters.

To register a namespace prefix:

1. From Setup, click **Create** > **Packages**.
2. Click **Edit**.

   Note: This button does not appear if you have already configured your developer settings.

3. Review the selections necessary to configure developer settings and click **Continue**.
4. Enter the namespace prefix you want to register.
5. Click **Check Availability** to determine if it is already in use.
6. Repeat if the namespace prefix you entered is not available.
7. Click **Review My Selections**.
8. Click **Save**.

Your namespace is used as a prefix to the components and Apex classes you are creating. In addition, use the namespace to address any apps you create by accessing:

`https://<mySalesforceInstance>.lightning.force.com/<namespace>/<appName>.app`, where `<mySalesforceInstance>` is the name of the instance hosting your org; for example, `na1`.

Enable Lightning Components

You must opt in to enable Lightning components for your organization.

1. From Setup, click **Develop** > **Lightning Components**.
2. Select the `Enable Lightning Components` checkbox.

   Warning: You can't use Force.com Canvas apps in Salesforce1 if you enable Lightning components. Any Force.com Canvas apps in your organization will no longer work in Salesforce1 if you enable Lightning components.

3. Click **Save**.

Create an Expense Object

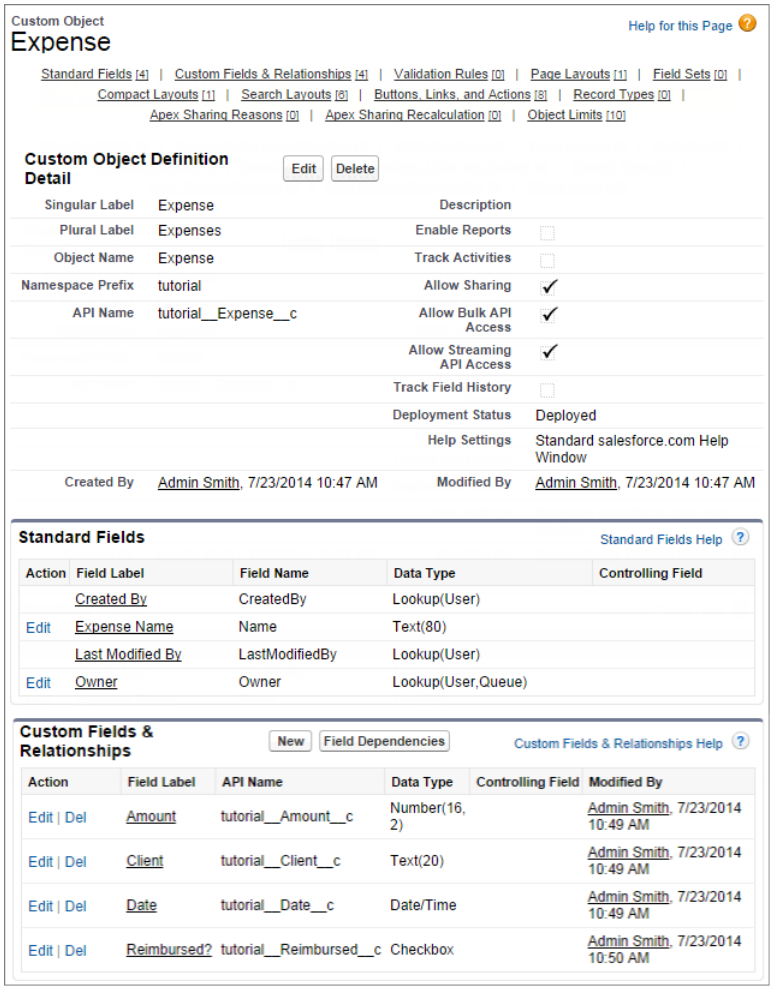
Create an expense object to store your expense records and data for the app.

1. From Setup, click **Create** > **Objects**.
2. Click **New Custom Object**.
3. Fill in the custom object definition.
   - For the *`Label`*, enter *`Expense`*.
   - For the *`Plural Label`*, enter *`Expenses`*
4. Click **Save** to finish creating your new object. The Expense detail page is displayed.

   Check that the API Name for your object is *`namespace__Expense__c`*, where `namespace` corresponds to your registered namespace prefix.
5. On the Expense detail page, add the following custom fields.

Field Type	Field Label
Number(16, 2)	Amount
Text (20)	Client
Date/Time	Date
Checkbox	Reimbursed?

When you finish creating the custom object, your Expense definition detail page should look similar to this.



6. Create a custom object tab to display your expense records.
   a. From Setup, click **Create** > **Tabs**.
   b. In the Custom Object Tabs related list, click **New** to launch the New Custom Tab wizard.
      - For the *Object*, select *Expense*.
      - For the *Tab Style*, click the lookup icon and select the *Credit Card* icon.
   c. Accept the remaining defaults and click **Next**.
   d. Click **Next** and **Save** to finish creating the tab.

      You should now see a tab for your Expenses at the top of the screen.

7. Create a few expense records.
   a. Click the Expenses tab and click **New**.
   b. Enter the values for these fields and repeat for the second record.

Expense Name	Amount	Client	Date	Reimbursed?
Lunch	20		4/1/2014 12:00 PM	Unchecked

Expense Name	Amount	Client	Date	Reimbursed?
Dinner	40.80	ABC Co.	4/2/2014 7:00 PM	Checked

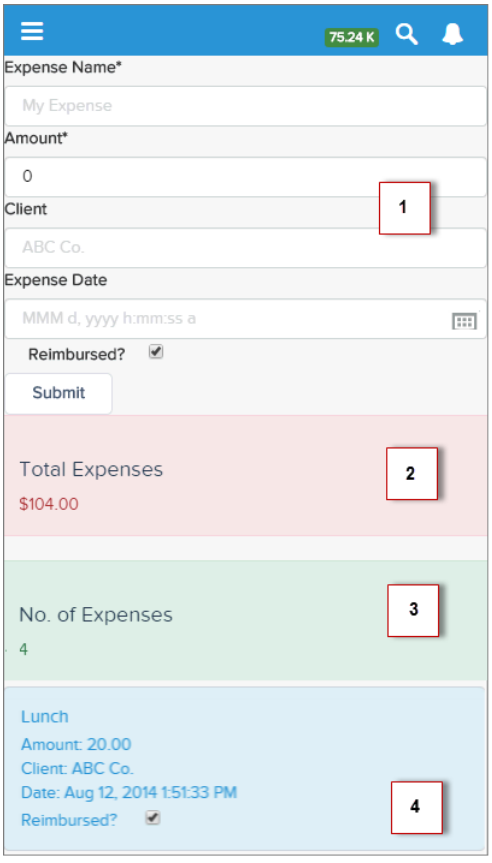
Create A Standalone Lightning App

This tutorial walks you through creating a simple expense tracker app using the Developer Console.

The goal of the app is to take advantage of many of the out-of-the-box Lightning components, and to demonstrate the client and server interactions using JavaScript and Apex. As you build the app, you'll learn how to use expressions to interact with data dynamically and use events to communicate data between components.

Make sure you've created the expense custom object shown in Create an Expense Object on page 8. Using a custom object to store your expense data, you'll learn how an app interacts with records, how to handle user interactions using client-side controller actions, and how to persist data updates using an Apex controller.

After you create the app, include it in Salesforce1 by following the steps in Adding Lightning Components to Salesforce1 on page 42. For packaging and distributing your apps on AppExchange, see Distributing Applications and Components on page 130.



1. The form contains Lightning input components (1) that update the view and expense records when the **Submit** button is pressed.
2. Counters are initialized (2) with total amount of expenses and number of expenses, and updated on record creation or deletion.

3. Display of expense list (3) uses Lightning output components and are updated as more expenses are added.
4. User interaction on the expense list (4) triggers an update event that saves the record changes.

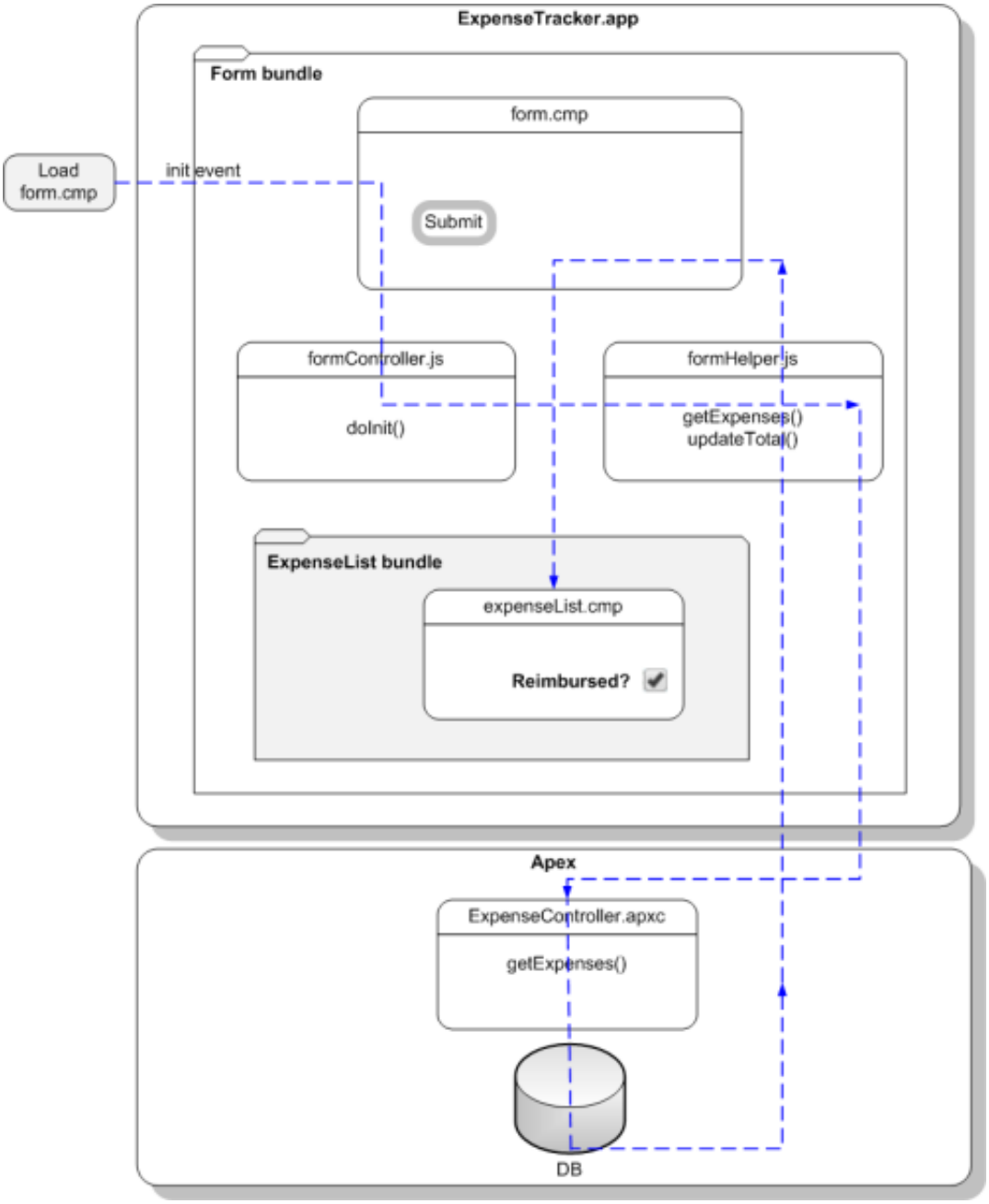
These are the resources you are creating for the expense tracker app.

Resources	Description
expenseTracker Bundle	
expenseTracker.app	The top-level component that contains all other components
expenseTracker.css	The styles for the app
Form Bundle	
form.cmp	A collection of Lightning input components to collect user input
formController.js	A client-side controller containing actions to handle user interactions on the form
formHelper.js	A client-side helper functions called by the controller actions
form.css	The styles for the form component
expenseList Bundle	
expenseList.cmp	A collection of Lightning output components to display data from expense records
expenseListController.js	A client-side controller containing actions to handle user interactions on the display of the expense list
expenseList.css	The styles for the display of the expense list
Apex Class	
ExpenseController.apxc	Apex controller that loads data, inserts, or updates an expense record
Event	
updateExpenseItem.evt	The event fired when an expense item is updated from the display of the expense list

Step 1: Create A Static Mockup

Create a static mockup in a `.app` file, which is the entry point for your app. It can contain other components and HTML markup.

The following flowchart summarizes the data flow in the app. The app retrieves data from the records through a combination of client-side controller and helper functions, and an Apex controller, which you'll create later in this quick start.



1. Click **Your Name** > **Developer Console** to open the Developer Console.
2. In the Developer Console, click **File** > **New** > **Lightning Application**.
3. Enter *expenseTracker* for the `Name` field in the New Lightning Bundle popup window. This creates a new app, `expenseTracker.app`.
4. In the source code editor, enter this code.

```
<aura:application>
     <link href='/resource/bootstrap/' rel="stylesheet"/>
    <div class="container">
        <h1>Add Expense</h1>
```

```
    </div>
</aura:application>
```

An application is a top-level component and the main entry point to your components. It can include components and HTML markup, such as `<div>` and `<header>` tags.

Note: The filepath name `bootstrap` corresponds to the name of the static resource that you will upload later in this quick start. Don't worry about the missing resource errors in the browser console as we won't be using those resources in this quick start.

5. Add styles to your new component.
   a. In the sidebar, click **STYLE**. This creates a new CSS resource, `expenseTracker.css`.
   b. Enter these CSS rule set.

```
.THIS h1 {
    background: url(/img/icon/creditCard32.png) no-repeat;
    padding-left: 40px;
}
```

Note: `THIS` is a keyword that adds namespacing to CSS to prevent any conflicts with another component's styling. The background icon refers to the credit card icon for the tab you created in Create an Expense Object on page 8.

6. Download the Salesforce1 styled theme available at http://developer.salesforcefoundation.org/bootstrap-sf1/ and upload the `bootstrap.css` file as a static resource. This file is located in the `/dist/css` directory.
   a. From **Setup**, click **Developer** > **Static Resources**. Click **New**.
   b. Enter *`bootstrap`* in the `Name` field. Click the **Choose File** button and select the `bootstrap.css` file. We'll not be uploading the other files to simplify this tutorial.
   c. Click **Save**.

7. Save your changes and click **Preview** in the sidebar to preview your app. Alternatively, navigate to `https://<mySalesforceInstance>.lightning.force.com/<namespace>/expenseTracker.app`, where `<mySalesforceInstance>` is the name of the instance hosting your org; for example, `na1`. You should see a header and icon on a gray background.

SEE ALSO:

aura:application

Step 2: Create A Component for User Input

Components are the building blocks of an app. They can be wired up to an Apex controller class to load your data. The component you create in this step provides a form that takes in user input about an expense, such as expense amount and date.

1. Click **File** > **New** > **Lightning Component**.
2. Enter *`form`* for the `Name` field in the New Lightning Bundle popup window. This creates a new component, `form.cmp`.
3. In the source code editor, enter this code. Replace `namespace` with the name of your registered namespace.

```
<aura:component>
  <aura:attribute name="expenses" type="namespace.Expense__c[]"/>
  <aura:attribute name="newExpense" type="namespace.Expense__c"
```

```
            default="{ 'sobjectType': 'namespace__Expense__c',
                       'Name': '',
                       'namespace__Amount__c': 0,
                       'namespace__Client__c': '',
                       'namespace__Date__c': '',
                       'namespace__Reimbursed__c': false
                     }"/>
  <!-- Attributes for Expense Counters -->
  <aura:attribute name="total" type="Double" default="0.00" />
  <aura:attribute name="exp" type="Double" default="0" />

  <!-- Input Form using components -->
  <form>
    <fieldset>
      <ui:inputText aura:id="expname" label="Expense Name"
                    class="form-control"
                    value="{!v.newExpense.name}"
                    placeholder="My Expense" required="true"/>
      <ui:inputNumber aura:id="amount" label="Amount"
                    class="form-control"
                    value="{!v.newExpense.namespace__Amount__c}"
                    placeholder="20.80" required="true"/>
      <ui:inputText aura:id="client" label="Client"
                    class="form-control"
                    value="{!v.newExpense.namespace__Client__c}"
                    placeholder="ABC Co."/>
      <ui:inputDateTime aura:id="expdate" label="Expense Date"
                    class="form-control"
                    value="{!v.newExpense.namespace__Date__c}"
                    displayDatePicker="true"/>
      <ui:inputCheckbox aura:id="reimbursed" label="Reimbursed?"
                    value="{!v.newExpense.namespace__Reimbursed__c}"/>
      <ui:button label="Submit" press="{!c.createExpense}"/>
    </fieldset>
  </form>

  <!-- Expense Counters -->
  <div class="row">
    <!-- Change the counter color to red if total amount is more than 100 -->
    <div class="{!v.total >= 100 ? 'alert alert-danger' : 'alert alert-success'}">
      <h3>Total Expenses</h3>$<ui:outputNumber value="{!v.total}" format=".00"/>
    </div>
    <div class="alert alert-success">
      <h3>No. of Expenses</h3><ui:outputNumber value="{!v.exp}"/>
    </div>
  </div>

  <!-- Display expense records -->
  <div class="row">
    <aura:iteration items="{!v.expenses}" var="expense">
        <p>{!expense.name}, {!expense.namespace__Client__c},
{!expense.namespace__Amount__c}, {!expense.namespace__Date__c},
{!expense.namespace__Reimbursed__c}</p>
    </aura:iteration>
```

```
    </div>
</aura:component>
```

Components provide a rich set of attributes and browser event support. Attributes are typed fields that are set on a specific instance of a component, and can be referenced using an expression syntax. All `aura:attribute` tags have name and type values. For more information, see Supported aura:attribute Types on page 141.

The attributes and expressions here will become clearer as you build the app. `{!v.exp}` evaluates the number of expenses records and `{!v.total}` evaluates the total amount. `{!c.createExpense}` represents the client-side controller action that runs when the **Submit** button is clicked, which creates a new expense. The `press` event in `ui:button` enables you to wire up the action when the button is pressed.

The expression `{!v.expenses}` wires up the component to the expenses object. `var="expense"` denotes the name of the variable to use for each item inside the iteration. `{!expense.namespace_Client__c}` represents data binding to the client field in the expense object.

Note: The default value for `newExpense` of type `namespace__Expense__c` must be initialized with the correct fields, including `sobjectType`. Initializing the default value ensures that the expense is saved in the correct format.

4. Click **STYLE** in the sidebar to create a new resource named `form.css`. Enter these CSS rule sets.

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

.THIS .uiButton {
    margin-bottom: 20px;
}

.THIS .row {
    width: 100%;
    margin-bottom: 20px;
}

.THIS .blue {
    background: #d9edf2;
}

.THIS .white {
    background: #ffffff;
}

.THIS .uiInput.uiInputDateTime {
    position: relative;
}

.THIS .uiInputDateTime+.datePicker-openIcon {
    position: absolute;
    display: inline-block;
    left: 90%;
    top: 25px;
    background-position: center;
    padding: 2% 5%;
}
```

```
.THIS .uiInputDefaultError li {
  list-style: none;
}
```

Note: The `.uiInputDefaultError` selector styles the default error component when you add field validation in Step 5: Enable Input for New Expenses on page 22.

5. Add the component to the app. In `expenseTracker.app`, add the new component to the markup.

   Replace `namespace` with the name of your registered namespace.

```
<aura:application>
     <link href='/resource/bootstrap/' rel="stylesheet"/>
    <div class="container">
        <h1>Add Expense</h1>

        <!-- Add the component here -->
        <namespace:form />
    </div>
</aura:application>
```

6. Save your changes and click **Update Preview** in the sidebar to preview your app. Alternatively, reload your browser.

   Note: In this step, the component you created doesn't display any data since you haven't created the Apex controller class yet.

Good job! You created a component that provides an input form and view of your expenses.

Beyond the Basics

The Lightning Component framework comes with a set of out-of-the-box components that are organized into different namespaces: `aura` and `ui`. The `ui` namespace provides components typical of a UI framework. For example, `ui:inputText` correponds to a text field. The `aura` namespace includes many components for core framework functionality, like `aura:iteration` as used in this step.

SEE ALSO:

aura:component

aura:iteration

Component Body

Step 3: Load the Expense Data

Load expense data using an Apex controller class. Display this data via component attributes and update the counters dynamically.

Create the expense controller class.

1. Click **File** > **New** > **Apex Class** and enter *`ExpenseController`* in the **New Class** window. This creates a new Apex class, `ExpenseController.apxc`.
2. Enter this code.

```
public class ExpenseController {
    @AuraEnabled
    public static List<Expense__c> getExpenses() {
```

```
        return [SELECT id, name, amount__c, client__c, date__c,
                    reimbursed__c, createdDate FROM Expense__c];
    }
}
```

The `getExpenses()` method contains a SOQL query to return all expense records. Recall the syntax `{!v.expenses}` in `form.cmp`, which displays the result of the `getExpenses()` method in the component markup.

Note: For more information on using SOQL, see the *Force.com SOQL and SOSL Reference*.

`@AuraEnabled` enables client- and server-side access to the controller method. Server-side controllers must be static and all instances of a given component share one static controller. They can return or take in any types, such as a List or Map.

Note: For more information on server-side controllers, see Apex Server-Side Controller Overview on page 118.

3. In `form.cmp`, update the `aura:component` tag to include the `controller` attribute. Add an `init` handler to load your data on component initialization. Replace `namespace` with the name of your registered namespace.

```
<aura:component controller="namespace.ExpenseController">
<aura:handler name="init" value="{!this}" action="{!c.doInit}" />
  <!-- Other aura:attribute tags here -->
  <!-- Other code here -->
</aura:component>
```

On initialization, this event handler runs the `doInit` action that you're creating next. This `init` event is fired before component rendering.

4. Add the client-side controller action for the `init` handler. In the sidebar, click **CONTROLLER** to create a new resource, `formController.js`. Enter this code.

```
({
    doInit : function(component, event, helper) {
       //Update expense counters
       helper.getExpenses(component);
    },//Delimiter for future code
})
```

During component initialization, the expense counters should reflect the latest sum and total number of expenses, which you're adding next using a helper function, `getExpenses(component)`.

Note: A client-side controller handles events within a component and can take in three parameters: the component to which the controller belongs, the event that the action is handling, and the helper if it's used. A helper is a resource for storing code that you want to reuse in your component bundle, providing better code reusability and specialization. For more information about using client-side controllers and helpers, see Handling Events with Client-Side Controllers on page 68 and Sharing JavaScript Code in a Component Bundle on page 97.

5. Create the helper function to display the expense records and dynamically update the counters. Click **HELPER** to create a new resource, `formHelper.js` and enter this code. Replace `namespace` with the name of your registered namespace.

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

```
            self.updateTotal(component);
        });
        $A.enqueueAction(action);
    },
  updateTotal : function(component) {
      var expenses = component.get("v.expenses");
      var total = 0;
      for(var i=0; i<expenses.length; i++){
          var e = expenses[i];
          total += e.namespace__Amount__c;
      }
      //Update counters
      component.set("v.total", total);
      component.set("v.exp", expenses.length);
    },//Delimiter for future code

})
```

`component.get("c.getExpenses")` returns an instance of the server-side action. `action.setCallback()` passes in a function to be called after the server responds. In that function, the `action` object is passed back in. To illustrate that it's two different pointers to that object, we called it `a`. But it ends up being the same `action` object. In `updateTotal`, you are retrieving the expenses and summing up their amount values and length of expenses, setting those values on the `total` and `exp` attributes.

Note: `$A.enqueueAction(action)` adds the action to the queue. All the action calls are asynchronous and run in batches. For more information about server-side actions, see Calling a Server-Side Action on page 119.

6. Save your changes and reload your browser.

   You should see the expense records created in Create an Expense Object on page 8. The counters aren't working at this point as you'll be adding the programmatic logic later.

Your app now retrieves the expense object and displays its records as a list, iterated over by `aura:iteration`. The counters now reflect the total sum and number of expenses.

In this step, you created an Apex controller class to load expense data. `getExpenses()` returns the list of expense records. By default, the framework doesn't call any getters. To access a method, annotate the method with `@AuraEnabled`, which exposes the data in that method. Only methods that are annotated with `@AuraEnabled` in the controller class are accessible to the components.

Component markup that uses the `ExpenseController` class can display the expense name or id with the `{!expense.name}` or `{!expense.id}` expression, as shown in Step 2: Create A Component for User Input on page 13.

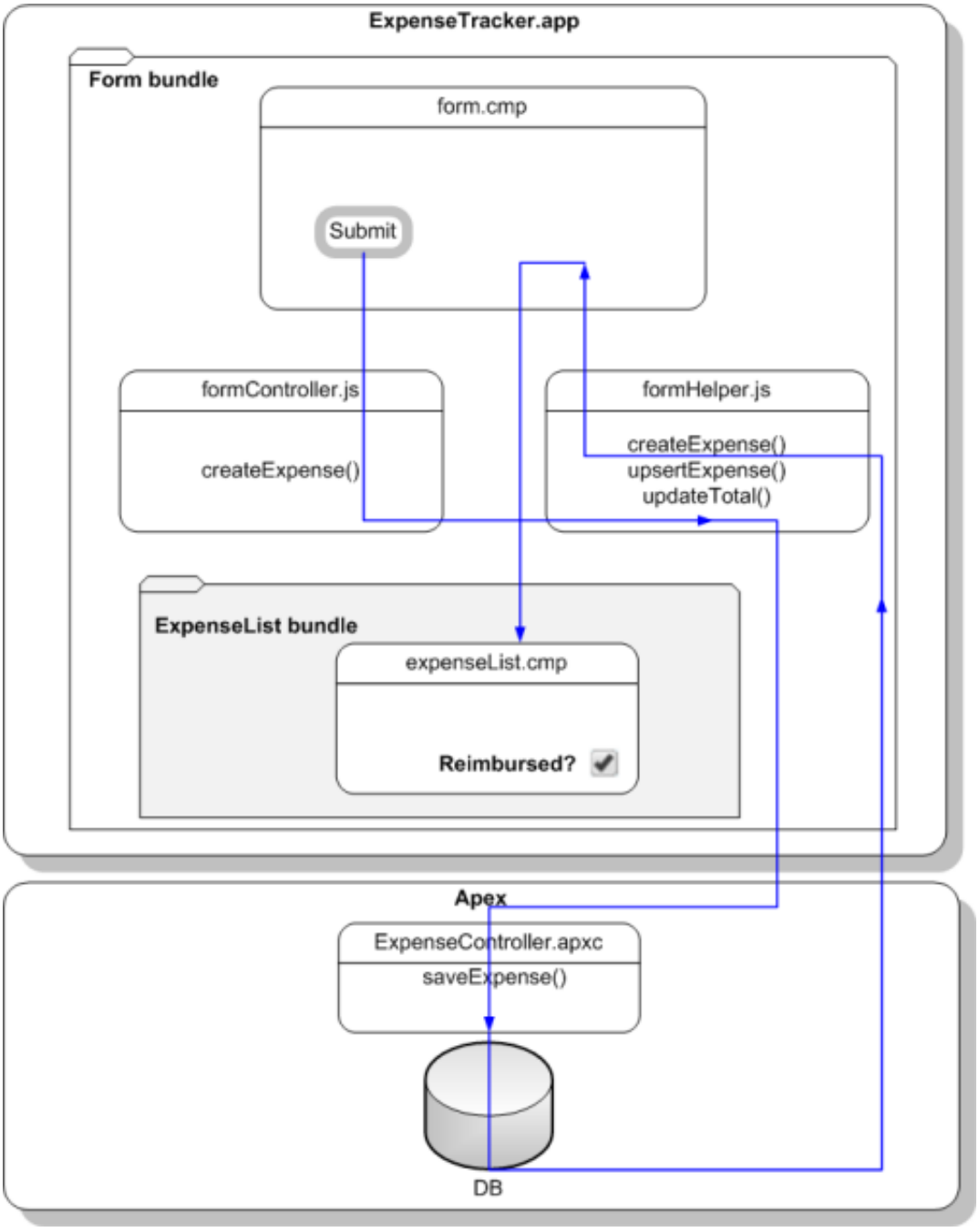
Beyond the Basics

Client-side controller definitions are surrounded by brackets and curly braces. The curly braces denotes a JSON object, and everything inside the object is a map of name-value pairs. For example, `updateTotal` is a name that corresponds to a client-side action, and the value is a function. The function is passed around in JavaScript like any other object.

Step 4: Create a Nested Component

As your component grows, you want to break it down to maintain granularity and encapsulation. This step walks you through creating a component with repeating data and whose attributes are passed to its parent component. You'll also add a client-side controller action to load your data on component initialization.

The following flowchart shows the flow of data in your app when you create a new expense. The data is captured when you click the **Submit** button in the component `form.cmp`, processed by your JavaScript code and sent to the server-side controller to be saved as a record. Data from the records is displayed in the nested component you are creating in this step.



1. Click **File** > **New** > **Lightning Component**.
2. Enter *`expenseList`* in the New Lightning Bundle window. This creates a new component, `expenseList.cmp`.
3. In `expenseList.cmp`, enter this code.

Replace `namespace` with the name of your registered namespace.

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

  <!-- Color the item blue if the expense is reimbursed -->
  <div class="{!v.expense.namespace__reimbursed__c == true
                ? 'listRecord recordLayout blue' : 'listRecord recordLayout white'}">

    <a aura:id="expense" href="{!'/' + v.expense.id}">
      <div class="itemTitle">{!v.expense.name}</div>
      <div class="recordItem">Amount:
          <ui:outputNumber
           value="{!v.expense.namespace__amount__c}" format=".00"/>
      </div>
      <div class="recordItem">Client:
          <ui:outputText
           value="{!v.expense.namespace__client__c}"/>
      </div>
      <div class="recordItem">Date:
          <ui:outputDateTime
           value="{!v.expense.namespace__date__c}" />
      </div>
      <div class="recordItem">Reimbursed?
          <ui:inputCheckbox
           value="{!v.expense.namespace__reimbursed__c}" click="{!c.update}"/>
      </div>
    </a>
  </div>
</aura:component>
```

Instead of using `{!expense.namespace__Amount__c}`, you're now using `{!v.expense.namespace__Amount__c}`. This expression accesses the `expense` object and the `amount` values on it. Note that only custom fields must be prefixed with namespaces. `expense` is of type `Expense__c`. For non-primitive types, use the format `myNamespace.myApexClass`.

Additionally, `href="{!'/' + v.expense.id}"` uses the expense ID to set the link to the detail page of each expense record.

4. Click **STYLE** in the sidebar and enter these CSS rule sets.

```
.THIS.recordLayout {
    list-style: none;
    padding: 14px;
    margin: 10px;
    border-bottom:1px solid #cfd4d9;
}

.THIS.listRecord {
    margin: 14px;
    border-radius: 5px;
    border: 1px solid #cfd4d9;
    position: relative;
}
```

```
.THIS .uiOutputText {
    line-height: 1.1em;
}

.THIS .itemTitle {
    font-size: 15px;
    padding-bottom: 3px;
    overflow: hidden;
    text-overflow: ellipsis;
    white-space: nowrap;
}

.THIS .recordItem {
    text-overflow: ellipsis;
    white-space: nowrap;
}

.THIS a:hover {
    text-decoration: none;
    background-color: #235636;
}
```

5. In `form.cmp`, update the `aura:iteration` tag to use the new nested component, `expenseList`. Locate the existing `aura:iteration` tag.

```
    <aura:iteration items="{!v.expenses}" var="expense">
        <p>{!expense.name}, {!expense.namespace__Client__c},
{!expense.namespace__Amount__c}, {!expense.namespace__Date__c},
{!expense.namespace__Reimbursed__c}</p>
    </aura:iteration>
```

   Replace it with an `aura:iteration` tag that uses the `expenseList` component. Replace `namespace` with the name of your registered namespace.

```
    <aura:iteration items="{!v.expenses}" var="expense">
      <namespace:expenseList expense="{!expense}"/>
    </aura:iteration>
```

   Notice how the markup is simpler as you're just passing each `expense` record to the `expenseList` component, which handles the display of the expense details.

6. Save your changes and reload your browser.

You created a nested component and pass its attributes to a parent component. Next, you'll learn how to process user input and update the expense object.

Beyond the Basics

When you create a component, you are providing the definition of that component. When you put the component in another component, you are create a reference to that component. This means that you can add multiple instances of the same component with different attributes. For more information about component attributes, see Component Composition on page 35.

SEE ALSO:

- Component Attributes
- aura:iteration
- Invoking Actions on Component Initialization

Step 5: Enable Input for New Expenses

When you enter text into the form and press Submit, you want to insert a new expense record. This action is wired up to the button component via the `press` attribute.

First, update the Apex controller with a new method that inserts or updates the records.

1. In the `ExpenseController` class, enter this code below the `getExpenses()` method.

```
@AuraEnabled
public static Expense__c saveExpense(Expense__c expense) {
    upsert expense;
    return expense;
}
```

The `saveExpenses()` method enables you to insert or update an expense record using the `upsert` operation.

Note: Fore more information about the `upsert` operation, see the *Apex Code Developer's Guide*.

2. Create the controller-side actions to create a new expense record when the **Submit** button is pressed. In `formController.js`, add this code after the `doInit` action.

```
createExpense : function(component, event, helper) {
    var amtField = component.find("amount");
    var amt = amtField.get("v.value");
    if (isNaN(amt)||amt==''){
        amtField.setValid("v.value", false);
        amtField.addErrors("v.value", [{message:"Enter an expense amount."}]);
    }
    else {
        amtField.setValid("v.value", true);
        var newExpense = component.get("v.newExpense");
        helper.createExpense(component, newExpense);
        }
},//Delimiter for future code
```

`createExpense` validates the name and amount fields using default error handling, which appends an error message represented by `ui:inputDefaultError`. The controller invalidates the input value using `setValid(false)` and clears any errors using `setValid(true)`. For more information on field validation, see Validating Fields on page 102.

Notice that you're passing in the arguments to a helper function `helper.createExpense()`, which then triggers the Apex class `saveExpense`.

Note: Recall that you specified the `aura:id` attributes in Step 2: Create A Component for User Input on page 13. `aura:id` enables you to find the component by name using the syntax `component.find("amount")` within the scope of this component and its controller.

3. Create the helper function to handle the record creation. Add these helper functions after the `updateTotal` function.

```
createExpense: function(component, expense) {
      this.upsertExpense(component, expense, function(a) {
        var expenses = component.get("v.expenses");
        expenses.push(a.getReturnValue());
        component.set("v.expenses", expenses);
        this.updateTotal(component);
      });
},
  upsertExpense : function(component, expense, callback) {
      var action = component.get("c.saveExpense");
      action.setParams({
          "expense": expense
      });
      if (callback) {
          action.setCallback(this, callback);
      }
      $A.enqueueAction(action);
    }
```

`createExpense` calls `upsertExpense`, which defines an instance of the `saveExpense` server-side action and sets the `expense` object as a parameter. The callback is executed after the server-side action returns, which updates the records, view, and counters. `$A.enqueueAction(action)` adds the server-side action to the queue of actions to be executed.

Note: Different possible action states are available and you can customize their behaviors in your callback. For more information on action callbacks, see Calling a Server-Side Action.

4. Save your changes and reload your browser. Test your app by entering `Breakfast, 10, ABC Co., Apr 30, 2014 9:00:00 AM`. For the date field, you can also use the date picker to set a date and time value. Click the Submit button. The record is added to both your component view and records, and the counters are updated.

Note: To debug your Apex code, use the Logs tab in the Developer Console. For example, if you don't have input validation for the date time field and entered an invalid date time format, you might get an INVALID_TYPE_ON_FIELD_IN_RECORD exception, which is listed both on the Logs tab in the Developer Console and in the response header on your browser. Otherwise, you might see an Apex error displayed in your browser. For more information on debugging your JavaScript code, see Debugging JavaScript Code on page 132.

Congratulations! You have successfully created a simple expense tracker app that includes several components, client- and server-side controllers, and helper functions. Your app now accepts user input, which updates the view and database. The counters are also dynamically updated as you enter new user input. The next step shows you how to add a layer of interactivity using events.

SEE ALSO:

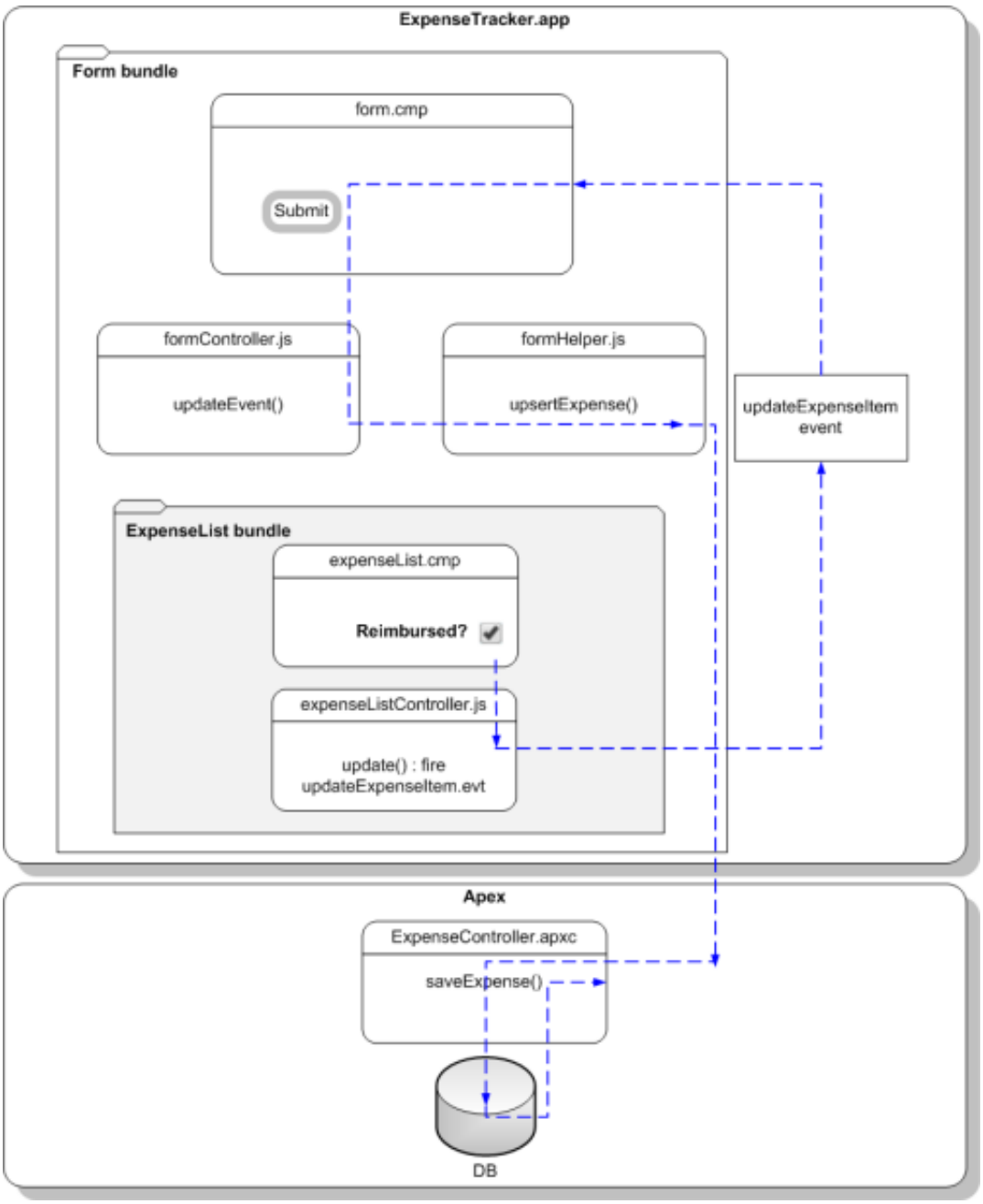
Handling Events with Client-Side Controllers

Calling a Server-Side Action

Step 6: Make the App Interactive With Events

Events add an interactive layer to your app by enabling you to share data between components. When the checkbox is checked or unchecked in the expense list view, you want to fire an event that updates both the view and records based on the relevant component data.

This flowchart shows the data flow in the app when a data change is captured by the selecting and deselecting of a checkbox on the `expenseList` component. When the **Reimbursed?** checkbox is selected or deselected, this browser click event fires the application event you're creating here. This event communicates the expense object to the handler component, and its controller calls the Apex controller method to update the relevant expense record, after which the response is ignored by the client since we won't be handling this server response here.



Let's start by creating the event and its handler before firing it and handling the event in the parent component.

1. Click **File** > **New** > **Lightning Event**.
2. Enter *updateExpenseItem* in the New Event window. This creates a new event, `updateExpenseItem.evt`.
3. In `updateExpenseItem.evt`, enter this code.

   The attribute you're defining in the event is passed from the firing component to the handlers.

   Replace `namespace` with the name of your registered namespace.

   ```
   <aura:event type="APPLICATION">
       <aura:attribute name="expense" type="namespace.Expense__c"/>
   </aura:event>
   ```

   The framework has two types of events: component events and application events. An application event is used here, which when fired notifies its handlers. In this case, `form.cmp` is notified and handles the event.

   Recall that `expenseList.cmp` contains the checkbox that's wired up to a client-side controller action, denoted by `change="{!c.update}"`. You'll set up the `update` action next.

4. In the `expenseList` sidebar, click **CONTROLLER**. This creates a new resource, `expenseListController.js`. Enter this code.

   Replace `namespace` with the name of your registered namespace.

   ```
   ({
       update: function(component, evt, helper) {
         var expense = component.get("v.expense");
         var updateEvent = $A.get("e.namespace:updateExpenseItem");
         updateEvent.setParams({ "expense": expense }).fire();
       }
   })
   ```

   When the checkbox is checked or unchecked, the `update` action runs, setting the `reimbursed` parameter value to `true` or `false`. The `updateExpenseItem.evt` event is fired with the updated `expense` object .

5. In the handler component, `form.cmp`, add this handler code before the `<aura:attribute>` tags.

   Replace `namespace` with the name of your registered namespace.

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

   This event handler runs the `updateEvent` action when the application event you created is fired.

6. Wire up the `updateEvent` action to handle the event. In `formController.js`, enter this code after the `createExpense` controller action.

   ```
   updateEvent : function(component, event, helper) {
       helper.upsertExpense(component, event.getParam("expense"));
   }
   ```

   This action calls a helper function and passes in `event.getParam("expense")`, which contains the expense object with its parameters and values in this format: `{ Name : "Lunch" , namespace__Client__c : "ABC  Co." , namespace__Reimbursed__c : true , CreatedDate :  "2014-08-12T20:53:09.000Z" , namespace__Amount__c : 20}`.

That's it! You have successfully added a layer of interaction in your expense tracker app using an application event. When you change the reimbursed status on the view, the `update` event is fired, handled by the parent component, which then updates the expense record by running the server-side controller action `saveExpense`.

Beyond the Basics

The framework fires several events during the rendering lifecycle, such as the `init` event you used in this tutorial. For example, you can also customize the app behavior during the `waiting` event when the client is waiting for a server response and when the `doneWaiting` event is fired to signal that the response has been received. This example shows how you can add text in the app during the `waiting` event, and remove it when the `doneWaiting` event is fired.

```
<!-- form.cmp markup -->
<aura:handler event="aura:waiting" action="{!c.waiting}"/>
<aura:handler event="aura:doneWaiting" action="{!c.doneWaiting}"/>
<aura:attribute name="wait" type="String"/>

<div class="wait">
    {!v.wait}
</div>
```

```
/** formController.js **/
waiting : function(component, event, helper) {
    component.set("v.wait", "updating...");
},
doneWaiting : function(component, event, helper) {
    component.set("v.wait", "");
}
```

The app displays this text when you click the **Submit** button to create a new record or when you click the checkbox on an expense item. For more information, see Events Fired During the Rendering Lifecycle on page 85.

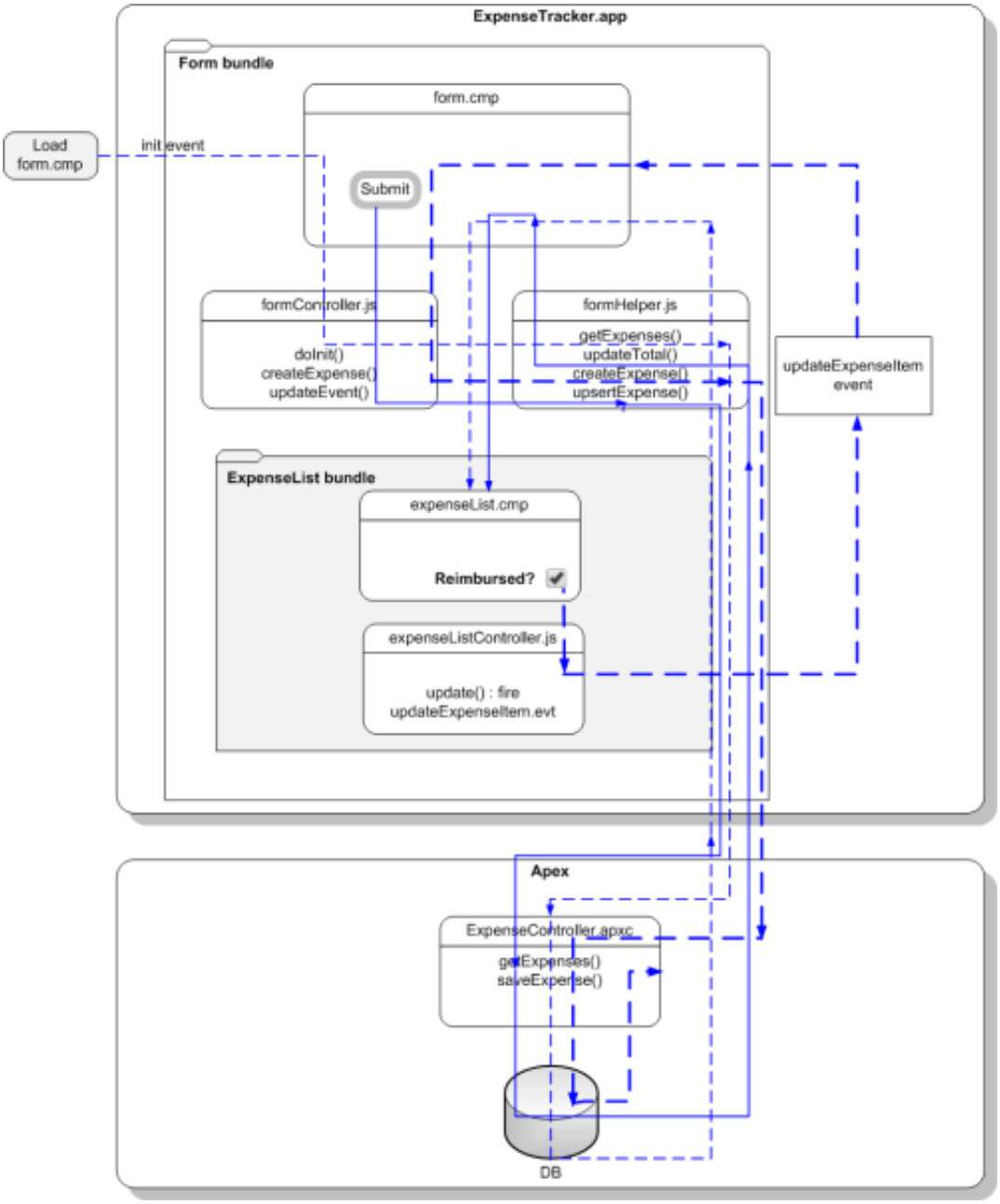
The app you just created is currently accessible as a standalone app by accessing `https://<mySalesforceInstance>.lightning.force.com/<namespace>/expenseTracker.app`, where `<mySalesforceInstance>` is the name of the instance hosting your org; for example, `na1`. To make it accessible in Salesforce1, see Adding Lightning Components to Salesforce1 on page 42. To package and distribute your app on AppExchange, see Distributing Applications and Components on page 130.

SEE ALSO:

- Application Events
- Event Handling Lifecycle

Summary

You created several components with controllers and events that interact with your expense records. The expense tracker app performs three distinct tasks: load the expense data and counters on app initialization, take in user input to create a new record and update the view, and handle user interactions by communicating relevant component data via events.



When `form.cmp` is initialized, the `init` handler triggers the `doInit` client-side controller, which calls the `getExpenses` helper function. `getExpenses` calls the `getExpenses` server-side controller to load the expenses. The callback sets the expenses data on the `v.expenses` attribute and calls `updateTotal` to update the counters.

Clicking the **Submit** button triggers the `createExpense` client-side controller. After field validation, the `createExpense` helper function is run, in which the `upsertExpense` helper function calls the `saveExpense` server-side controller to save the record. The callback pushes the new expense to the list of expenses and updates the attribute `v.expenses` in `form.cmp`, which in turn updates the expenses in `expenseList.cmp`. Finally, the helper calls `updateTotal` to update the counters represented by the `v.total` and `v.exp` attributes.

`expenseList.cmp` displays the list of expenses. When the **Reimbursed?** checkbox is selected or deselected, the `click` event triggers the `update` client-side controller. The`updateExpenseItem` event is fired with the relevant expense passed in as a

parameter. `form.cmp` handles the event, triggering the `updateEvent` client-side controller. This controller action then calls the `upsertExpense` helper function, which calls the `saveExpense` server-side controller to save the relevant record.

CHAPTER 3 Components

In this chapter ...

Components are the functional units of the Lightning Component framework.

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

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

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

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

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

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

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

Component Namespace

Every component is part of a namespace, which is used to group related components together.

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

Note: You must register a namespace to use the Lightning Component framework in your organization. See Before You Begin on page 7.

Component Bundles

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

Resource	Resource Name	Usage	See Also
Component or Application	`sample.cmp` or `sample.app`	The only required resource in a bundle. Contains markup for the component or app. Each bundle contains only one component or app resource.	Components on page 29 aura:application on page 134
CSS Styles	`sample.css`	Styles for the component.	CSS in Components on page 32

Resource	Resource Name	Usage	See Also
Controller	`sampleController.js`	Client-side controller methods to handle events in the component.	Handling Events with Client-Side Controllers on page 68
Documentation	`sample.auradoc`	A description, sample code, and one or multiple references to example components	Providing Component Documentation on page 43
Renderer	`sampleRenderer.js`	Client-side renderer to override default rendering for a component.	Client-Side Rendering to the DOM on page 99
Helper	`sampleHelper.js`	JavaScript functions that can be called from any JavaScript code in a component's bundle	Sharing JavaScript Code in a Component Bundle on page 97

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 unique within a component and is only scoped to the component.

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

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

Find the button component by calling `cmp.find("button1")`, where `cmp` is a reference to the component containing the button.

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

For more information, see the JavaScript API at `https://<mySalesforceInstance>.lightning.force.com/auradocs/reference.app`, where `<mySalesforceInstance>` is the name of the instance hosting your org; for example, `na1`.

Global IDs

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

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

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

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

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

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

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

SEE ALSO:

Finding Components by ID

HTML in Components

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

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

Warning: Some tags, like `<applet>` and `<font>`, aren't supported. For a full list of unsupported tags, see Supported HTML Tags on page 140.

Unescaping HTML

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

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

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

SEE ALSO:

Supported HTML Tags

CSS in Components

CSS in Components

Style your components with CSS.

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

For external CSS resources, see Styling Apps on page 92.

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

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

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

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

SEE ALSO:

- Adding and Removing Styles
- HTML in Components

Component Attributes

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

Use the `<aura:attribute>` tag in a component's markup to add an attribute to the component. Let's look at a sample component, `helloAttributes.cmp`:

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

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

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

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

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

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

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

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

Warning: This only works for attributes of type String.

Expressions

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

`{!<expression>}` is 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.

Attribute Validation

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

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

SEE ALSO:

- Supported aura:attribute Types
- Expressions

Component Composition

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

Let's see how we can fit components together.

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

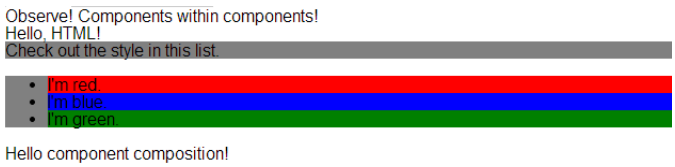
Component source

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

    <auradocs:helloHTML/>

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

Output



Including an existing component is similar to including an HTML tag: we just reference the component by its "descriptor", which is of the form <namespace>:<component>. `nestedComponents.cmp` references the `helloHTML.cmp` component, which lives in the `auradocs` namespace. Hence, its descriptor is `auradocs:helloHTML`.

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

Here is the source for `helloHTML.cmp`.

Component source

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

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

  <ul>
    <li class="red">I'm red.</li>
    <li class="blue">I'm blue.</li>
```

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

CSS source

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

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

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

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

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

Output

Here is the source for `helloAttributes.cmp`.

Component source

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

Attribute Passing

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

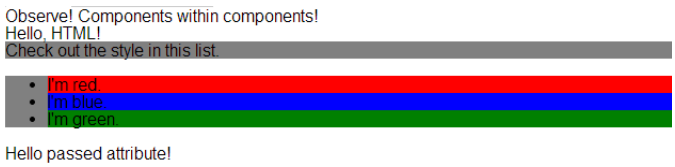
Component source

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

    <auradocs:helloHTML/>
```

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

Output



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

Definitions versus Instances

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

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

Component source

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

    <auradocs:helloHTML/>

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

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

Output

Component Body

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

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

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

Setting the Body Content

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

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

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

This is a shortcut for:

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

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

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

This is a shortcut for:

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

Accessing the Component Body

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

SEE ALSO:

- aura:set
- Working with a Component Body in JavaScript

Component Facets

A facet is any attribute of type `Aura.Component[]`.

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

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

Component source

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

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

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

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

Component source

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

    <auradocs:facetHeader>

        Nice body!

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

</aura:component>
```

SEE ALSO:

Component Body

Using Labels

The framework supports labels to enable you to separate field labels from your code.

IN THIS SECTION:

Input Component Labels

A label describes the purpose of an input component. To set a label on an input component, use the `label` attribute.

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.

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:" value="54" />
```

The label is placed on the left of the input field and can be hidden by setting `labelClass="assistiveText"`. `assistiveText` is a global style class used to support accessibility.

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

Localization

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

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

Component source

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

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

Additionally, you can use the global value provider, `$Locale`, to obtain a browser's locale information. By default, 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.

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);
```

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

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

SEE ALSO:

Global Value Providers

Enabling Lightning Components

You must opt in to enable Lightning components for your organization.

1. From Setup, click **Develop** > **Lightning Components**.
2. Select the `Enable Lightning Components` checkbox.

Warning: You can't use Force.com Canvas apps in Salesforce1 if you enable Lightning components. Any Force.com Canvas apps in your organization will no longer work in Salesforce1 if you enable Lightning components.

3. Click **Save**.

Adding Lightning Components to Salesforce1

Make your Lightning components available for Salesforce1 users.

In the component you wish to add, you must include `implements="force:appHostable"` in your `aura:component` tag and save your changes.

EDITIONS

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

USER PERMISSIONS

To create Lightning Component Tabs:
- "Customize Application"

```
<aura:component implements="force:appHostable">
```

The `appHostable` interface makes the component available on the navigation menu in Salesforce1.

Include your components in the Salesforce1 navigation menu by following these steps.

1. Create a custom tab for this component.
   a. From Setup, click **Create** > **Tabs**.
   b. Click **New** in the Lightning Component Tabs related list.
   c. Select the Lightning component to display in the custom tab.
   d. Enter a label to display on the tab.
   e. Select the tab style and click **Next**.
   f. When prompted to add the tab to profiles, accept the default and click **Save**.

   Note: Creating a custom tab is a prerequisite to enabling your component in the Salesforce1 navigation menu, but accessing your Lightning component from the full Salesforce site is not supported.

2. Include your Lightning component in the Salesforce1 navigation menu.
   a. From Setup, click **Mobile Administration** > **Mobile Navigation**.
   b. Select the custom tab you just created and click **Add**.
   c. Sort items by selecting them and clicking **Up** or **Down**.

   In the navigation menu, items appear in the order you specified. The first item in the Selected list becomes your users' Salesforce1 landing page.

3. Check your output by going to the Salesforce1 mobile browser app. Your new menu item should appear in the navigation menu.

   Note: By default, the mobile browser app is turned on for your organization. For more information on using the Salesforce1 mobile browser app, see the *Salesforce1 App Developer Guide*.

Adding 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: For all the out-of-the-box components, see the `Components` folder at `https://<mySalesforceInstance>.lightning.force.com/auradocs/reference.app`, where `<mySalesforceInstance>` is the name of the instance hosting your org; for example, `na1`. 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.

To add a new custom component to your app, see Using the Developer Console on page 4.

SEE ALSO:

Component Composition
Component Attributes
Events

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, click **DOCUMENTATION** in the component sidebar of the Developer Console. The following example shows the DocDef for `np:myComponent`.

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

```
<aura:documentation>
    <aura:description>
        <p>An <code>np:myComponent</code> component represents an element that executes
an action defined by a controller.</p>
        <!--More markup here, such as <pre> for code samples-->
    </aura:description>
    <aura:example name="myComponentExample" ref="np:myComponentExample" label="Using the
np:myComponent Component">
```

```
 <p>This example shows a simple setup of <code>myComponent</code>.</p>
    </aura:example>
    <aura:example name="mySecondExample" ref="np:mySecondExample" label="Customizing the
np:myComponent Component">
        <p>This example shows how you can customize <code>myComponent</code>.</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 `<aura:component>`.
`<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="myComponentExample" ref="np:myComponentExample" label="Using the
np:myComponent Component">
```

The following is an example component that demonstrates how `np:myComponent` can be used.

```
<!--The np:myComponentExample example component-->
<aura:component>
    <np:myComponent>
        <aura:set attribute=”myAttribute”>This sets the attribute on the np:myComponent
component.</aura:set>
        <!--More markup that demonstrates the usage of np:myComponent-->
    </np:myComponent>
</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"/>`
`<aura:registerEvent>`	`<aura:registerEvent name="keydown" type="ui:keydown" description="Indicates that a key is pressed"/>`

Viewing the Documentation

The documentation you create will be available at `https://<mySalesforceInstance>.lightning.force.com/auradocs/reference.app`, where `<mySalesforceInstance>` is the name of the instance hosting your org; for example, `na1`.

SEE ALSO:

Reference Overview

CHAPTER 4 Expressions

In this chapter ...

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 (integer, string, and so on), a boolean, a JavaScript object, a component or collection, a controller method such as an action method, and other useful results.

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

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

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.

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.

SEE ALSO:

Example Expressions

Example Expressions

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

Dynamic Output

The simplest way to use expressions is to simply output them. Values used in the expression can be from component attributes, literal values, booleans, and so on.

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

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

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

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

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

Passing Values

Use expressions to pass values around. For example:

```
<aura:iteration items="{!v.expenses}" var="expense">
```

The `{!v.expenses}` expression passes the `expenses` attribute to the `aura:iteration` tag. The expression is not evaluated yet. When the `aura:iteration` tag renders, it evaluates the expression to retrieve the `items` value.

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

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

Conditional Expressions

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

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

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

```
<aura: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>
```

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

SEE ALSO:

Value Providers

Handling Events with Client-Side Controllers

Value Providers

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

The most common value providers are `v` and `c`, as in view and controller.

Value Provider	Description
`v`	A component's attribute set
`c`	A component's controller with actions and event handlers for the component

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

Values in a value provider are accessed as named properties. To use a value, separate the value provider and the property name with a dot (period). For example, `v.body`.

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

Accessing Fields and Related Objects

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

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

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

SEE ALSO:

Example Expressions

Global Value Providers

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

The global value providers are:

- `globalID`—See Component IDs on page 31.
- `$Browser`—See `$Browser` on page 49.
- `$Locale`—See `$Locale` on page 49.

$Browser

The `$Browser` global value provider provides 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 how to get some `$Browser` attributes.

Component source

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

$Locale

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

Attribute	Description	Sample Value
`language`	Returns the language code.	"en", "de", "zh"

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

Example: This example shows how to get some `$Locale` attributes.

Component source

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

The framework also provides localization support for input and output components.

SEE ALSO:

Localization

Expression Evaluation

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

Operators are a subset of those available in JavaScript, and evaluation order and precedence are generally the same as JavaScript. Parentheses enable you to ensure a specific evaluation order. What you may find surprising about expressions is how often they are evaluated. The simplistic answer is, as often as they need to be. A more complete answer is that the framework can notice when things change, and trigger re-rendering of any components that are affected. Dependencies are handled automatically. This is one of the fundamental benefits of the 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 Example Expressions on page 47.

Expression Operators Reference

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

Arithmetic Operators

Expressions based on arithmetic operators result in numerical values.

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

Numeric Literals

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

String Operators

Expressions based on string operators result in string values.

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

String Literals

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

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

Comparison Operators

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

Operator	Alternative	Usage	Description

`==`	`eq`	`1 == 1`
		`1 == 1.0`
		`1 eq 1`
			Returns `true` if the operands are equal. This comparison is valid for all data types.

`!=`	`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 `<` 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 `<=` to use it in component markup. Alternatively, you can use the `le` operator.
`>=`	`ge`	`42 >= 42` `42 ge 42`	Returns `true` if the first operand is numerically greater than or equal to the second.

Logical Operators

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

Operator	Usage	Description
`&&`	`isEnabled && hasPermission`	Returns `true` if both operands are individually true. You must escape the `&&` operator to `&&` 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

Operator Usage	Description
	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.	-

Function	Alternative	Usage	Description	Corresponding Operator
`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.	`+`

Array Functions

Function	Alternative	Usage	Description	Corresponding Operator
`length`		`myArray.length`	Returns the length of the array.	

Comparison Functions

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

Function	Usage	Description	Corresponding Operator
`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.

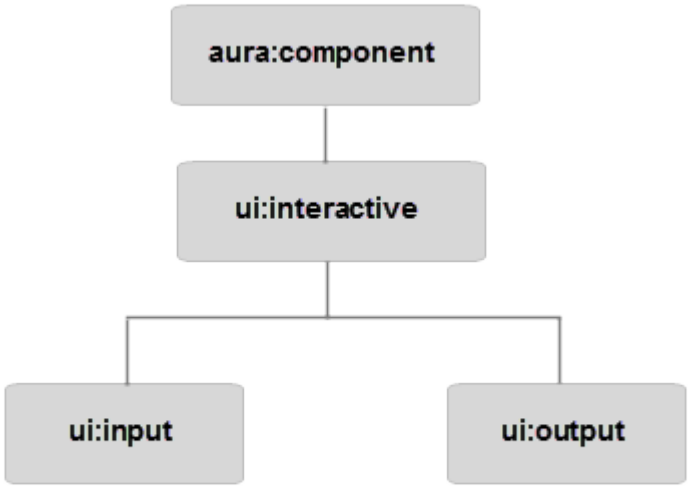
Function	Usage	Description	Corresponding Operator
`and`	`and(isEnabled, hasPermission)`	Returns `true` if both arguments are true.	`&&`
`or`	`or(hasPermission, hasVIPPass)`	Returns `true` if either one of the arguments is true.	`||`
`not`	`not(isNew)`	Returns `true` if the argument is false.	`!`

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)

CHAPTER 5 User Interface Overview

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. Interactive user interface components such as `ui:input` and `ui:output` extend `ui:interactive`, which provides common user interface events like keyboard and mouse interactions. Each component can be styled accordingly. For all the components available, see the component reference at `https://<mySalesforceInstance>.lightning.force.com/auradocs/reference.app`, where `<mySalesforceInstance>` is the name of the instance hosting your org; for example, `na1`.



SEE ALSO:

- Input Components Overview
- Components
- Component Bundles

Input Components Overview

Users interact with your app through input elements to select or enter values. The framework provides a range of input elements such as text fields, buttons, checkboxes, and so on.

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

Using the Input Components

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

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

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

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

These are the common field components you can use.

Field Type	Description	Related Components

Date and time	An input field for entering date and time.	`ui:inputDate`
		`ui:inputDateTime`
		`ui:outputDate`
		`ui:outputDateTime`

Number	An input field for entering a numerical value.	`ui:inputNumber` `ui:outputNumber`
Text	An input field for entering single line of text.	`ui:inputText` `ui:outputText`

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

  ```
  <!-- Component markup -->
  <ui:button label="Find" labelClass="assistiveText" class="img" />

  /** CSS **/
  THIS.uiButton.img {
      background: url(/path/to/img) no-repeat;
      width:50px;
  ```

```
    height:25px;
}
```

The `assistiveText` class hides the label from view but makes it available to assistive technologies.

- Button with Text and Image

```
<!-- Component markup -->
<ui:button label="Find" />

/** CSS **/
THIS.uiButton {
    background: url(/path/to/img) no-repeat;
}
```

HTML Rendering

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

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

Working with Click Events

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

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

Styling Your Buttons

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

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

SEE ALSO:

- Handling Events with Client-Side Controllers
- CSS in Components

Date and Time Fields

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

Date and Time fields are represented by the following components.

Field Type	Description	Related Components
Date	An input field for entering a date of type `text`.	`ui:inputDate` `ui:outputDate`
Date and Time	An input field for entering a date and time of type `text`.	`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">
  <label class="uiLabel-left uiLabel">
    <span>Birthday</span>
  </label>
  <input placeholder="MMM d, yyyy" type="text" class="uiInput uiInputDate">
  <a class="datePicker-openIcon" aria-haspopup="true">
    <span class="assistiveText">Date Picker</span>
  </a>
  <div class="uiDatePicker">
    <!--Date picker set to visible when icon is clicked-->
  </div>
</div>
```

Localizing the Date and Time

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

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

Styling Your Date and Time Fields

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

To render the output for the `ui:inputNumber` component, use `ui:outputNumber`.

Using the Number Fields

This example shows a number field, which displays a value of `10`.

```
<aura:attribute name="num" type="integer" default="10"/>
<ui:inputNumber aura:id="num" label="Age" value="{!v.num}"/>
```

The previous example results in the following HTML.

```
<div class="uiInput uiInputText uiInputNumber">
  <label class="uiLabel-left uiLabel">
    <span>Enter age</span>
  </label>
  <input aria-describedby placeholder type="text"
         class="uiInput uiInputText uiInputNumber">
</div>
```

Returning a Valid Number

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

This example returns `100,000`.

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

This example also returns `100,000`.

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

Formatting and Localizing the Number Fields

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

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

Styling Your Number Fields

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

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

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

SEE ALSO:

- Input Component Labels
- Handling Events with Client-Side Controllers
- Localization
- CSS in Components

Text Fields

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

Text fields are represented by the following components.

Field Type	Description	Related Components
Email	An input field for entering an email address.	`ui:inputEmail` `ui:outputEmail`
Phone	An input field for entering a phone number.	`ui:inputPhone` `ui:outputPhone`
Text	An input field for entering a single-line text.	`ui:inputText` `ui:outputText`

Using the Text Fields

This is a basic set up of an email field.

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

This example results in the following HTML.

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

Styling Your Text Fields

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

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

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

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:

- Input Component Labels
- Handling Events with Client-Side Controllers
- Localization
- CSS in Components

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"`. This example sets the inital value of the checkbox.

```
<aura:attribute name="check" type="Boolean" default="true"/>
<ui:inputcheckbox value="{!v.check}"/>
```

Disabled State

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

The previous example results in the following HTML.

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

Working with Events

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

The following code crosses out the checkbox item.

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

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

Styling Your Checkboxes

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

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

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

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

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

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

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

SEE ALSO:

- Handling Events with Client-Side Controllers
- CSS in Components

Field-level Errors

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

SEE ALSO:

- Handling Events with Client-Side Controllers
- CSS in Components

COMMUNICATING WITH EVENTS

CHAPTER 6 Events

In this chapter ...

If you have ever developed with JavaScript or Java Swing, you should be familiar with the idea of event-driven programming. You write handlers that respond to interface events as they occur. The events may or may not have been triggered by user interaction.

In the Lightning Component framework, events are fired from JavaScript controller actions. Events can contain attributes that can be set before the event is fired and read when the event is handled.

Events are declared by the `aura:event` tag in a `.evt` resource, and they can have one of two types: component or application. The event type is set by either `type="COMPONENT"` or `type="APPLICATION"` in the `aura:event` tag.

Handling Events with Client-Side Controllers

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

Each action function takes in three parameters: the component to which the controller belongs, the event that the action is handling, and the helper if it's used. Client-side controllers are surrounded by brackets and curly braces to denote a JSON object containing a map of name-value pairs.

Creating a Client-Side Controller

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

To create a client-side controller using the Developer Console, click **CONTROLLER** in the sidebar of the component.

Calling Client-Side Controller Actions

Let's start by looking at events on different implementations of an HTML tag. The following example component creates three different buttons, of which only the last two works. 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/>
    <input type="button" value="Hybrid HTML Button" onclick="{!c.handleClick}"/>
    <br/>
    <ui:button label="Framework Button" press="{!c.handleClick}"/>
    <br/>
    {!v.text}
</aura:component>
```

Client-side controller source

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

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

```
    }
}
```

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

If you know some JavaScript, you might be tempted to write something like the first "Flawed" button because you know that HTML tags are first-class citizens in the framework. However, the "Flawed" button won't work though as the framework has its own event system. DOM events are mapped to Lightning events, since HTML tags are mapped to Lightning components.

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.

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

In this simple scenario, there is little functional difference between working with the "Framework" button or the "Hybrid" HTML button. However, 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.

Accessing Component Attributes

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

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

Invoking Another Action in the Controller

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

SEE ALSO:

- Sharing JavaScript Code in a Component Bundle
- Event Handling Lifecycle
- Invoking Actions on Component Initialization
- Creating Server-Side Logic with Controllers

Component Events

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

Create Custom Component Event

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

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

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

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

Register Component Event

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

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

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

Fire Component Event

To get a reference to a component event in JavaScript, use `getEvent("evtName")` where `evtName` matches the `name` attribute in `<aura:registerEvent>`. Use `fire()` to fire the event from an instance of a component. For example, in an action function in a client-side controller:

```
var compEvent = cmp.getEvent("sampleComponentEvent");
// set some data for the event (also known as event shape)
// compEvent.setParams(...);
compEvent.fire();
```

Component Handling Its Own Event

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

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

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

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

Handle Component Event of Instantiated Component

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

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

When you instantiate `<auradocs:eventsNotifier>` in another component, use the value of the `name` attribute from the `<aura:registerEvent>` tag to register the handler. For example, if an `<auradocs:eventsHandler>` component includes `<auradocs:eventsNotifier>` in its markup, `eventsHandler` instantiates `eventsNotifier` and can handle any events thrown by `eventsNotifier`. Here's how `<auradocs:eventsHandler>` instantiates `<auradocs:eventsNotifier>`:

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

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

Handle Component Event Dynamically

A component can have its handler bound dynamically via JavaScript. This is useful if a component is created in JavaScript on the client-side. See Dynamically Adding Event Handlers on page 110.

Get the Source of a Component Event

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

SEE ALSO:

Application Events
Handling Events with Client-Side Controllers
Advanced Events Example

Component Event Example

Here's a simple use case of using a component event to update an attribute in another component.

1. A user clicks a button in the notifier component, `ceNotifier.cmp`.
2. The client-side controller for `ceNotifier.cmp` sets a message in a component event and fires the event.
3. The handler component, `ceHandler.cmp`, contains the notifier component, and handles the fired event.
4. The client-side controller for `ceHandler.cmp` sets an attribute in `ceHandler.cmp` based on the data sent in the event.

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

Component Event

ceEvent.evt

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

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

Notifier Component

ceNotifier.cmp

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

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

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

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

ceNotifierController.js

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

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

Handler Component

ceHandler.cmp

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

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

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

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

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

</aura:component>
```

ceHandlerController.js

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

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

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

Put It All Together

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

`http://<mySalesforceInstance>/<namespace>/docsample/ceHandler.cmp`, where `mySalesforceInstance` is the name of the instance hosting your org; for example, `na1.salesforce.com`.

If you want to access data on the server, you could extend this example to call a server-side controller from the handler's client-side controller.

SEE ALSO:

- Component Events
- Creating Server-Side Logic with Controllers
- Application Event Example

Application Events

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

Create Custom Application Event

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

Use `type="APPLICATION"` in the `<aura:event>` tag for an application event. For example, this is an application event with one `message` attribute.

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

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

Register Application Event

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

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

Fire Application Event

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

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

Handle Application Event

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

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

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

Get the Source of an Application Event

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

Events Fired on 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 85.

SEE ALSO:

Component Events
Handling Events with Client-Side Controllers
Advanced Events Example

Application Event Example

Here's a simple use case of using an application event to update an attribute in another component.

1. A user clicks a button in the notifier component, `aeNotifier.cmp`.
2. The client-side controller for `aeNotifier.cmp` sets a message in a component event and fires the event.
3. The handler component, `aeHandler.cmp`, handles the fired event.
4. The client-side controller for `aeHandler.cmp` sets an attribute in `aeHandler.cmp` based on the data sent in the event.

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

Application Event

aeEvent.evt

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

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

Notifier Component

aeNotifier.cmp

The notifier component uses `aura:registerEvent` to declare that it may fire the application event. Note that the `name` attribute is required but not used for application events. The `name` attribute is only relevant for component events.

The button in the component contains a `press` browser event that is wired to the `fireApplicationEvent` action in the client-side controller. Clicking this button invokes the action.

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

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

```
    </p>
</aura:component>
```

aeNotifierController.js

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

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

Handler Component

aeHandler.cmp

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

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

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

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

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

aeHandlerController.js

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

```
{
    handleApplicationEvent : function(cmp, event) {
        var message = event.getParam("message");

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

Container Component

aeContainer.cmp

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

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

Put It All Together

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

`http://<mySalesforceInstance>/<namespace>/docsample/aeContainer.cmp`, where `mySalesforceInstance` is the name of the instance hosting your org; for example, `na1.salesforce.com`.

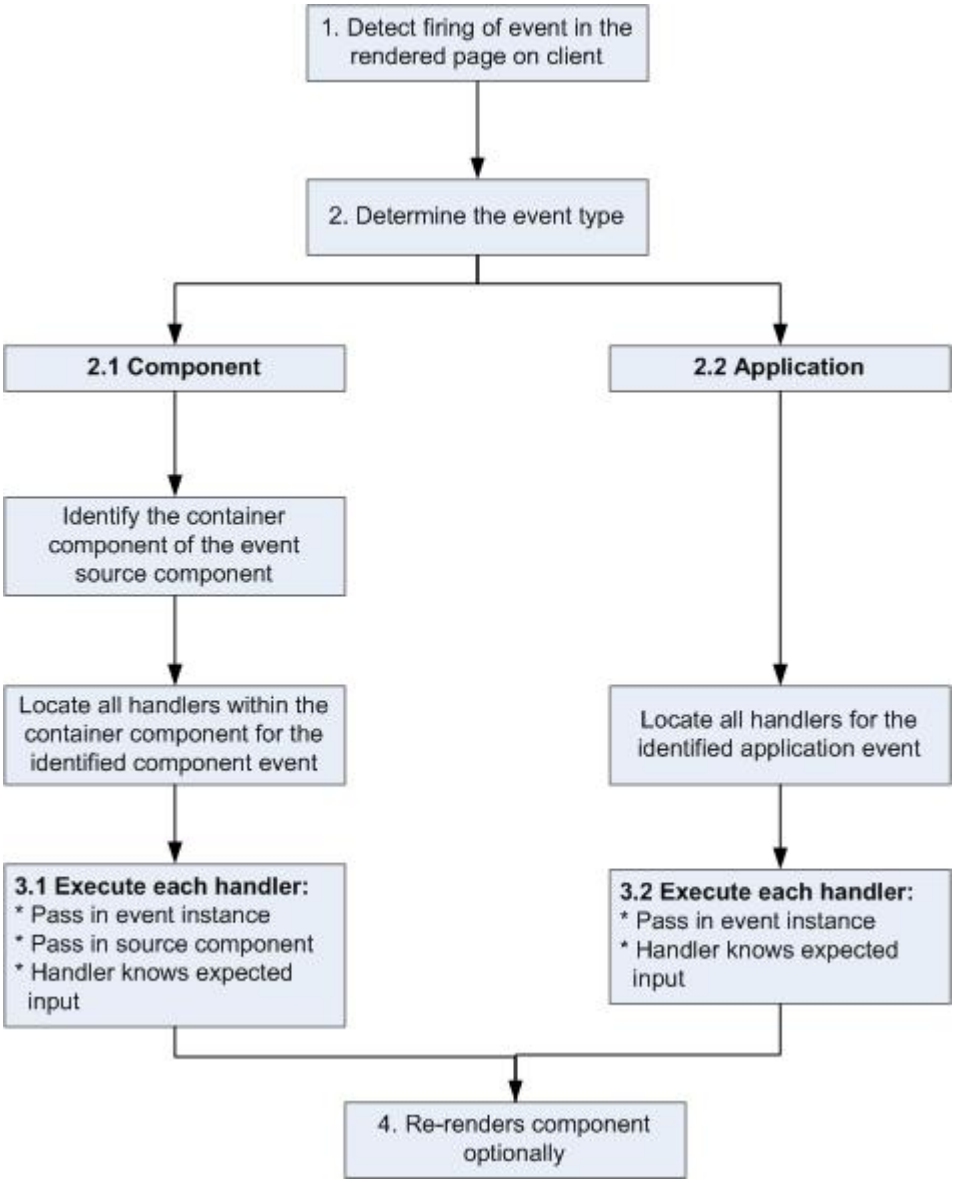
If you want to access data on the server, you could extend this example to call a server-side controller from the handler's client-side controller.

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

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

compEvent.evt

```
<aura:event type="COMPONENT">
    <!-- pass context of where the event was fired to the handler. -->
    <aura:attribute name="context" type="String"/>
</aura:event>
```

Application Event

appEvent.evt

```
<aura:event type="APPLICATION">
    <!-- pass context of where the event was fired to the handler. -->
    <aura:attribute name="context" type="String"/>
</aura:event>
```

Notifier Component

eventsNotifier.cmp

The notifier component contains a `press` browser event to initiate a component or application event.

The notifier uses `aura:registerEvent` tags to declare that it may fire the component and application events. Note that the `name` attribute is required but left empty for the application event.

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

Component source

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

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

CSS source

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

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

Client-side controller source

The controller fires the event.

```
{
    fireComponentEvent : function(cmp, event) {
        var parentName = cmp.get("v.parentName");

        // Look up event by name, not by type
        var compEvents = cmp.getEvent("componentEventFired");

        compEvents.setParams({ "context" : parentName });
        compEvents.fire();
    },

    fireApplicationEvent : function(cmp, event) {
        var parentName = cmp.get("v.parentName");

        // note different syntax for getting application event
        var appEvent = $A.get("e.auradocs:appEvent");

        appEvent.setParams({ "context" : parentName });
        appEvent.fire();
    }
}
```

You can click the buttons to fire component and application events but there is no change to the output because we haven't wired up the handler component to react to the events yet.

The controller sets the `context` attribute of the component or application event to the `parentName` of the notifier component before firing the event. We will see how this affects the output when we look at the handler component.

Handler Component

eventsHandler.cmp

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

Component source

```
<aura:component>
  <aura:attribute name="name" type="String"/>
  <aura:attribute name="mostRecentEvent" type="String" default="Most recent event handled:"/>

  <aura:attribute name="numComponentEventsHandled" type="Integer" default="0"/>
  <aura:attribute name="numApplicationEventsHandled" type="Integer" default="0"/>
  <aura:handler event="auradocs:appEvent" action="{!c.handleApplicationEventFired}"/>

  <div>
    <h3>This is {!v.name}</h3>
    <p>{!v.mostRecentEvent}</p>
    <p># component events handled: {!v.numComponentEventsHandled}</p>
```

```
    <p># application events handled: {!v.numApplicationEventsHandled}</p>
    <auradocs:eventsNotifier parentName="{!v.name}"
componentEventFired="{!c.handleComponentEventFired}"/>
  </div>
</aura:component>
```

CSS source

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

Client-side controller source

```
{
    handleComponentEventFired : function(cmp, event) {
        var context = event.getParam("context");
        cmp.set("v.mostRecentEvent",
            "Most recent event handled: COMPONENT event, from " + context);

        var numComponentEventsHandled =
            parseInt(cmp.get("v.numComponentEventsHandled")) + 1;
        cmp.set("v.numComponentEventsHandled", numComponentEventsHandled);
    },

    handleApplicationEventFired : function(cmp, event) {
        var context = event.getParam("context");
        cmp.set("v.mostRecentEvent",
            "Most recent event handled: APPLICATION event, from " + context);

        var numApplicationEventsHandled =
            parseInt(cmp.get("v.numApplicationEventsHandled")) + 1;
        cmp.set("v.numApplicationEventsHandled", numApplicationEventsHandled);
    }
}
```

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

You can click buttons and the UI now changes to indicate the type of event. The click count increments to indicate whether it's a component or application event. We aren't finished yet though. Notice that the source of the event is undefined as the event `context` attribute hasn't been set .

Container Component

eventsContainer.cmp

Component source

```
<aura:component>
    <auradocs:eventsHandler name="eventsHandler1"/>
    <auradocs:eventsHandler name="eventsHandler2"/>
</aura:component>
```

The container component contains two handler components. It sets the `name` attribute of both handler components, which is passed through to set the `parentName` attribute of the notifier components. This fills in the gaps in the UI text that we saw when we looked at the notifier or handler components directly.

Click the **Click here to fire a component event** button for either of the event handlers. Notice that the **# component events handled** counter only increments for that component because only the firing component's handler is notified.

Click the **Click here to fire an application event** button for either of the event handlers. Notice that the **# application events handled** counter increments for both the components this time because all the handling components are notified.

SEE ALSO:

- Component Event Example
- Application Event Example
- Event Handling Lifecycle

Firing Lightning Events from Non-Lightning Code

You can fire Lightning events from JavaScript code outside a Lightning app. For example, your Lightning app might need to call out to some non-Lightning code, and then have that code communicate back to your Lightning 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 Lightning app. Let's call this event `mynamespace:externalEvent`. You'll fire this event when your non-Lightning code is done by including this JavaScript in your non-Lightning 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 Lightning app is loaded.

SEE ALSO:

- Application Events
- Modifying Components from External JavaScript

Events Best Practices

Here are some best practices for working with events.

Separate Low-Level Events from Business Logic Events

It's a good practice to handle low-level events, such as a click, in your event handler and refire them as higher-level events, such as an `approvalChange` event or whatever is appropriate for your business logic.

Dynamic Actions based on Component State

If you need to invoke a different action on a click event depending on the state of the component, try this approach:

1. Store the component state as a discrete value, such as New or Pending, in a component attribute.
2. Put logic in your client-side controller to determine the next action to take.
3. If you need to reuse the logic in your component bundle, put the logic in the helper.

For example:

1. Your component markup contains `<ui:button label="do something" press="{!c.click}" />`.
2. In your controller, define the `click` function, which delegates to the appropriate helper function or potentially fires the correct event.

Using a Dispatcher Component to Listen and Relay Events

If you have a large number of handler component instances listening for an event, it may be better to identify a dispatcher component to listen for the event. The dispatcher component can perform some logic to decide which component instances should receive further information and fire another component or application event targeted at those component instances.

SEE ALSO:

Handling Events with Client-Side Controllers

Events Anti-Patterns

Events Anti-Patterns

These are some anti-patterns that you should avoid when using events.

Don't Fire an Event in a Renderer

Firing an event in a renderer can cause an infinite rendering loop.

Don't do this!

```
afterRender: function(cmp, helper) {
    this.superAfterRender();
    $A.get("e.myns:mycmp").fire();
}
```

Instead, use the `init` hook to run a controller action after component construction but before rendering. Add this code to your component:

```
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
```

For more details, see .Invoking Actions on Component Initialization on page 107.

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

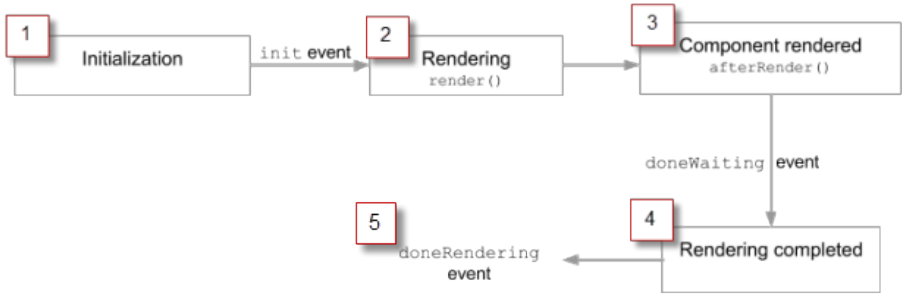
A component is instantiated, rendered, and rerendered during its lifecycle. A component is rerendered only when there's a programmatic or value change that would require a rerender, such as when a browser event triggers an action that updates its data.

The component lifecycle starts when the client sends an HTTP request to the server and the component configuration data is returned to the client. No server trip is made if the component definition is already on the client from a previous request and the component has no server dependencies.

Before going into the rendering lifecycle on the client, it's useful to understand the server-side and client-side processing for component requests in brief. The framework builds the component definition and all its dependencies in the server, including definitions for interfaces, controllers, and actions.. After creating a component instance, the serialized component definitions and instances are sent down to the client. Definitions are cached but not the instance data.

The client deserializes the response to create the JavaScript objects or maps, resulting in an instance tree used to render the component instance. The client locates the custom renderer in the component bundle or uses the default renderer method.

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



1. The `init` event is fired by the component service that constructs the components to signal that initialization has completed.

   ```
   <aura:handler name="init" value="{!this}" action="{!.c.doInit}"/>
   ```

   You can customize the init handler and add your own controller logic. For more information, see Invoking Actions on Component Initialization on page 107.

2. `render()` is called to start component rendering. The renderer for `aura:component` has a base implementation of `render()`, but your component can override this method in a custom renderer. For more information, see Client-Side Rendering to the DOM on page 99.

3. `afterRender()` is called to signal that rendering is completed for each of these component definitions. It enables you to interact with the DOM tree after the framework rendering service has inserted DOM elements.
4. To indicate that the client is done waiting for a response to the server request XHR, the `doneWaiting` event is fired. You can handle this event by adding a handler wired to a client-side controller action.
5. The framework checks whether any components need to be rerendered and rerenders any "dirtied" components to reflect any updates to attribute values, for example. Finally, the `doneRendering` event is fired the end of the rendering lifecycle.

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

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

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

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

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
`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

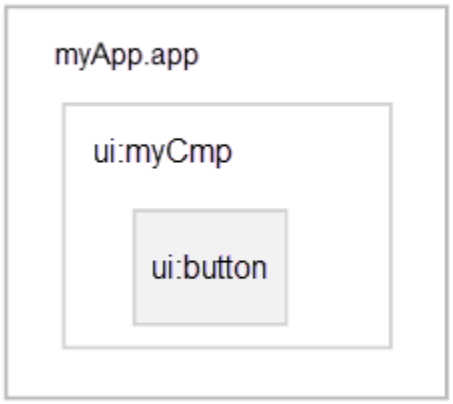
When rendering is done, this example calls `afterRender()` eight times for these component definitions. The `doneWaiting` event is fired, followed by the `doneRendering` event.

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 `ui:myCmp` with a `ui:button` component.



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

Customizing the `doneWaiting` Handler

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

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

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

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

Customizing the `doneRendering` Handler

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

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

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

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

```
({
  doneRendering: function(cmp, event, helper) {
    if(!cmp.get("v.isDoneRendering")){
      //do something after app is first rendered
    }
  }
})
```

SEE ALSO:

Client-Side Rendering to the DOM

CHAPTER 7 App Basics

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` resource.

On a production server, the `.app` resource is the only addressable unit in a browser URL. Access an app using the URL:

`https://<mySalesforceInstance>.lightning.force.com/<namespace>/<appName>.app`, where `<mySalesforceInstance>` is the name of the instance hosting your org; for example, `na1`.

SEE ALSO:

aura:application

Supported HTML Tags

Designing App UI

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

Let's take a look at the `accounts.app` resource created in Create A Standalone Lightning App.

```
<aura:application>
    <h1>Accounts</h1>
    <div class="container">
        <!-- Other components or markup here -->
    </div>
    </aura:application>
```

`accounts.app` contains HTML tags and component markup. You can use HTML tags like `<div class="container">` to design the layout of your app.

SEE ALSO:

aura:application

Content Security Policy Overview

The framework uses Content Security Policy (CSP) to control the source of content that can be loaded on a page.

CSP is a Candidate Recommendation of the W3C working group on Web Application Security. The framework uses the `Content-Security-Policy` HTTP header recommended by the W3C.

The framework's CSP covers these resources:

JavaScript Libraries
All JavaScript libraries must be uploaded to Salesforce static resources. For more information, see Accessing JavaScript Libraries in Markup on page 91.

HTTPS Connections for Resources
All external fonts, images, frames, and CSS must use an HTTPS URL.

Browser Support

CSP is not enforced for all browsers. For a list of browsers that enforce CSP, see `caniuse.com`.

Finding CSP Violations

Any policy violations are logged in the browser's developer console. The violations look like this:

```
Refused to load the script 'https://externaljs.docsample.com/externalLib.js'
because it violates the following Content Security Policy directive: ...
```

If your app's functionality is not affected, you can ignore the CSP violation.

Requesting CSP Exceptions

If your app is not working due to a CSP violation, contact Salesforce to request a CSP exception for your org. Include the violation message from your browser's developer console in any communication.

Accessing JavaScript Libraries in Markup

To reference a JavaScript library that you've uploaded as a static resource, use a `<script>` tag in your `.app` resource:

```
<script src="/resource/resourceName" type="text/javascript"></script>
```

`resourceName` is the `Name` of the static resource. Note that the framework doesn't currently support the `$Resource` global variable available in Visualforce.

For more information on static resources, see "What is a Static Resource?" in the Salesforce online help.

CHAPTER 8 Styling Apps

An app is a special top-level component whose markup is in a `.app` resource. Just like any other component, you can put CSS in its bundle in a resource called `<appName>.css`.

For example, if the app markup is in `notes.app`, its CSS is in `notes.css`.

External CSS Resources

To use an external CSS resource, add a `<link>` tag within your `<aura:application>` resource.

```
<link rel="stylesheet" type="text/css"
href="https://netdna.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css"/>
```

Note: The framework's content security policy mandates that external CSS resources must use an HTTPS connection. For more information, see .Content Security Policy Overview on page 90.

If you have other components in your `<aura:application>` tag, those components don't inherit the styles from your external resources when they are included in Salesforce1. CSS declarations for components included in Salesforce1 should be added to the CSS resource in the component bundle.

SEE ALSO:

- CSS in Components
- Using JavaScript Libraries
- Adding Lightning Components to Salesforce1

Vendor Prefixes

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

You only need to write the unprefixed version, and the framework automatically adds any prefixes that are necessary when generating the CSS output. If you choose to add them, they are used as-is. This enables you to specify alternative values for certain prefixes.

Example: For example, this is an unprefixed version of `border-radius`.

```
.class {
  border-radius: 2px;
}
```

The previous declaration results in the following declarations.

```
.class {
  -webkit-border-radius: 2px;
  -moz-border-radius: 2px;
```

```
  border-radius: 2px;
}
```

CHAPTER 9 Using JavaScript

In this chapter ...

Use JavaScript for client-side code. The `Aura` object is the top-level object in the JavaScript framework code. For all the methods available in the `Aura` class, see the JavaScript API at `https://<mySalesforceInstance>.lightning.force.com/auradocs/reference.app`, where `<mySalesforceInstance>` is the name of the instance hosting your org; for example, `na1`.

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

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

Expressions in JavaScript Code

In JavaScript, use string syntax to evaluate an expression. For example, this expression retrieves the `label` attribute in a component.

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

Note: Only use the `{! }` expression syntax in markup in `.app` or `.cmp` resources.

Accessing the DOM

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

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 two very important guidelines for accessing the DOM from a component or app.

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

Using Renderers

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

Using Expressions

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

Using JavaScript Libraries

The framework's content security policy mandates that external JavaScript libraries must be uploaded to Salesforce static resources. For more information, see Content Security Policy Overview on page 90.

SEE ALSO:

aura:application

Working with Attribute Values in JavaScript

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

Example: In these examples, `cmp` is a reference to a component in your JavaScript code. It's usually easy to get a reference to a component in JavaScript code.

Get an Attribute Value

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

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

Set an Attribute Value

To set the value of a component's `label` attribute:

```
cmp.set("v.label","This is a label");
```

Get a Boolean Attribute Value

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

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

For example, the following attribute returns `true` when passed into `$A.util.getBooleanValue()`.

```
<aura:attribute name="myString" type="String" default="my string"/>
```

If the attribute is of type Boolean, `cmp.get("v.myBoolean")` returns the boolean value and `$A.util.getBooleanValue()` is not needed.

Validate that an Attribute Value is Defined

To determine if a component's `label` attribute is defined:

```
var isDefined = !$A.util.isUndefined(cmp.get("v.label"));
```

Validate that an Attribute Value is Empty

To determine if a component's `label` attribute is empty:

```
var isEmpty = $A.util.isEmpty(cmp.get("v.label"));
```

SEE ALSO:

Working with a Component Body in JavaScript

Working with a Component Body in JavaScript

These are useful and common patterns for working with a component's body in JavaScript.

Example: 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.

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 resource is part of the component bundle and is auto-wired via the naming convention, `<componentName>Helper.js`.

To create a helper using the Developer Console, click **HELPER** in the sidebar of the component. This helper file is valid for the scope of the component to which it's auto-wired.

Using a Helper in a Renderer

Add a helper argument to a renderer function to enable the function to use the helper. In the renderer, specify `(component, helper)` as parameters in a function signature to enable the function to access the component's helper. These are standard parameters and you don't have to access them in the function. The following code shows an example on how you can override the `afterRender()` function in the renderer and call `open` in the helper method.

detailsRenderer.js

```
({
    afterRender : function(component, helper){
        helper.open(component, null, "new");
    }
})
```

detailsHelper.js

```
({
    open : function(component, note, mode, sort){
        if(mode === "new") {
            //do something
        }
        // do something else, such as firing an event
    }
})
```

For an example on using helper methods to customize renderers, see Client-Side Rendering to the DOM.

Using a Helper in a Controller

Add a `helper` argument to a controller function to enable the function to use the helper. Specify `(component, event, helper)` in the controller. These are standard parameters and you don't have to access them in the function.

The following code shows you how to call the `updateItem` helper function in a controller, which can be used with a custom event handler.

```
({
    newItemEvent: function(component, event, helper) {
        helper.updateItem(component, event.getParam("item"));
    }
})
```

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

```
({
    updateItem : function(component,item, 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

Client-Side Rendering to the DOM

The framework's 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. 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.

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

Rendering Lifecycle

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

1. A browser event triggers one or more Lightning events.
2. Each Lightning event triggers one or more actions that can update data. The updated data can fire more events.
3. The rendering service tracks the stack of events that are fired.
4. When all the data updates from the events are processed, the framework rerenders all the components that own modified data.

For more information, see Events Fired During the Rendering Lifecycle .

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

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

Creating a Renderer

You don't normally have to write a custom renderer, but if you want to customize rendering behavior, you can create a client-side renderer in a component bundle. A renderer file is part of the component bundle and is auto-wired if you follow the naming convention, `<componentName>Renderer.js`. For example, the renderer for `sample.cmp` would be in `sampleRenderer.js`.

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

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

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

Customizing Component Rendering

Customize rendering by creating a `render()` function in your component's renderer to override the base `render()` function, which updates the DOM.

The `render()` function typically returns a DOM node, an array of DOM nodes, or nothing. The base HTML component expects DOM nodes when it renders a component.

You generally want to extend default rendering by calling `superRender()` from your `render()` function before you add your custom rendering code. Calling `superRender()` creates the DOM nodes specified in the markup.

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

- A renderer should only modify DOM elements that are part of the component. You should never break component encapsulation by reaching in to another component and changing its DOM elements, even if you are reaching in from the parent component.
- A renderer should never fire an event. An alternative is to use an `init` event instead.

Rerendering Components

When an event is fired, it may trigger actions to change data and call `rerender()` on affected components. The `rerender()` function enables components to update themselves based on updates to other components since they were last rendered. This function doesn't return a value.

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

You generally want to extend default rerendering by calling `superRerender()` from your `renderer()` function before you add your custom rerendering code. Calling `superRerender()` chains the rerendering to the components in the `body` attribute.

Accessing the DOM After Rendering

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

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

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

Unrendering Components

The base `unrender()` function deletes all the DOM nodes rendered by a component's `render()` function. It is called by the framework when a component is being destroyed. Customize this behavior by overriding `unrender()` in your component's renderer. This can be useful when you are working with third-party libraries that are not native to the framework.

You generally want to extend default unrendering by calling `superUnrender()` from your `unrender()` function before you add your custom code.

Ensuring Client-Side Rendering

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

Rendering Example

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

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

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

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

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

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

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

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

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

```
                if (disabled) {
                    element.setAttribute('disabled', 'disabled');
                } else {
                    element.removeAttribute('disabled');
                }
            }
        }
    }
}
```

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

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

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

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

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

SEE ALSO:

- Accessing the DOM
- Invoking Actions on Component Initialization
- Component Bundles
- Events
- Sharing JavaScript Code in a Component Bundle

Validating Fields

You can validate fields using JavaScript. Typically, you validate the user input, identify any errors, and display the error messages. You can use the framework's default error handling or customize it with your own error handlers.

Default Error Handling

The framework can handle and display errors using the default error component, `ui:inputDefaultError`, without using custom error handlers. The following example shows how the framework handles a validation error and uses the default error component to display the error message.

Component source

```
<aura:component>
    Enter a number: <ui:inputNumber aura:id="inputCmp"/> <br/>
    <ui:button label="Submit" press="{!c.doAction}"/>
</aura:component>
```

Client-side controller source

```
{
    doAction : function(component) {
        var inputCmp = component.find("inputCmp");
        var value = inputCmp.get("v.value");

        // is input numeric?
        if (isNaN(value)) {
            // set error
            inputCmp.setValid("v.value", false);
            inputCmp.addErrors("v.value", [{message:"Input not a number: " + value}]);
        } else {
                // clear error
                inputCmp.setValid("v.value", true);
        }
    }
}
```

When you enter a value and click **Submit**, an action in the controller validates the input and displays an error message if the input is not a number. Entering a valid input clears the error. The controller invalidates the input value using `setValid(false)` and clears any error using `setValid(true)`. You can add error messages to the input value using `addErrors()`.

SEE ALSO:

- Handling Events with Client-Side Controllers
- Component Events

Throwing Errors

The framework gives you flexibility in handling unrecoverable and recoverable app errors in JavaScript code.

Unrecoverable Errors

Use `$A.error("error message here")` for unrecoverable errors, such as an error that prevents your app from starting successfully. It shows a stack trace on the page.

Recoverable Errors

To handle recoverable errors, use a component, such as `ui:message` or `ui:dialog`, to tell the user about the problem.

This sample shows you the basics of throwing and catching an error in a JavaScript controller.

Component source

```
<aura:component>

    <br/>

    <p>Click the button to trigger the controller to throw an error.</p>

    <br/>
```

```
    <div aura:id="div1"></div>

    <ui:button label="Throw an Error" press="{!c.throwErrorForKicks}"/>

</aura:component>
```

Client-side controller source

```
({
    throwErrorForKicks: function(cmp) {
        // this sample always throws an error
        var hasPerm = false;
        try {
            if (!hasPerm) {
                throw new Error("You don't have permission to edit this record.");
            }
        }
        catch (e) {
            // config for a dynamic ui:message component
            var componentConfig = {
                componentDef : "markup://ui:message",
                attributes : {
                    values : {
                        title : "Sample Thrown Error",
                        severity : "error",
                        body : [
                            {
                                componentDef : "markup://ui:outputText",
                                attributes : {
                                    values : {
                                        value : e.message
                                    }
                                }
                            }
                        ]
                    }
                }
            };

            $A.componentService.newComponentAsync(
                this,
                function(message) {
                    var div1 = cmp.find("div1");

                    // Replace existing body with the dynamic component
                    div1.set("v.body", message);
                },
                componentConfig
            );



        }
    }
})
```

See the controller code for an example of throwing an error in a try-catch block. The message in the error is displayed to the user in a dynamically created `ui:message` component.

SEE ALSO:

Validating Fields

CHAPTER 10 JavaScript Cookbook

In this chapter ...

This section includes code snippets and samples that can be used in various JavaScript files.

Invoking Actions on Component Initialization

You can update a component or fire an event after component construction but before rendering.

Component source

```
<aura:component>
    <aura:attribute name="setMeOnInit" type="String" default="default value" />

    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <br/>

    <p>This value is set in the controller after the component initializes and before
rendering.</p>

    <br/>
    <p><b>{!v.setMeOnInit}</b></p>

</aura:component>
```

Client-side controller source

```
({
    doInit: function(cmp) {
       // Set the value. This is not a very interesting sample as it just sets an attribute

        // but you could fire an event here instead
        cmp.set("v.setMeOnInit", "controller init magic!");
    }
})
```

Let's look at the **Component source** to see how this works. The magic happens in this line.

```
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
```

This registers an `init` event 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.

Setting `value="{!this}"` marks this as a value event. You should always use this setting for an `init` event.

SEE ALSO:

- Handling Events with Client-Side Controllers
- Client-Side Rendering to the DOM
- Component Attributes
- Detecting Data Changes

Detecting Data Changes

Automatically firing an event

You can configure a component to automatically invoke a client-side controller action when a value in one of the component's attributes changes. When the value changes, the `valueChange.evt` event is automatically fired. The `valueChange.evt` is an event with `type="VALUE"` that takes in two attributes, `value` and `index`.

Manually firing an event

In contrast, other component and application events are fired manually by `event.fire()` in client-side controllers.

For example, in the component, define a handler with `name="change"`.

```
<aura:handler name="change" value="{!v.items}" action="{!c.itemsChange}"/>
```

A component can have multiple `<aura:handler name="change">` tags to detect changes to different attributes.

In addition to the `name` attribute, `aura:handler` includes the `value` and `action` attributes.

Attribute Name	Type	Description
value	Object	The value for which you want to detect changes.
action	Object	The client-side controller action that is run when a change is detected.

In the controller, define the action for the handler.

```
({
    itemsChange: function(cmp, evt) {
        var v = evt.getParam("value");
        if (v === cmp.get("v.items")) {
        //do something
        }
    }
})
```

When a change occurs to a value that is represented by the `change` handler, the framework handles the firing of the event and rerendering of the component. For an example of detecting data changes, see the `aura:iteration` component.

SEE ALSO:

Invoking Actions on Component Initialization

Finding Components by ID

You can retrieve a component by its ID in JavaScript code. For example, a component has a local ID of `button1`.

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

You can find the button 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.

You can also retrieve a component by its global ID if you already have a value for the component's `globalId` in your code.

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

SEE ALSO:

- Component IDs
- Value Providers

Dynamically Creating Components

You can create a component dynamically from your client-side JavaScript code using the `newComponentAsync()` method.

Note: The `newComponentAsync()` method replaces the deprecated `newComponent()` and `newComponentDeprecated()` methods.

`$A.componentService.newComponentAsync(callbackScope, callback, config, attributeValueProvider, localCreation, doForce, forceServer)` takes in a required `callback` function that returns your newly created component, and a required `config` object, which provides the component descriptor and attributes. Refer to the JavaScript API reference for a full description of all the arguments.

This sample code creates a new `ui:button` component with the local ID, attaches an event handler to the new button, and appends the button to the body.

```
createButton : function(cmp) {
    $A.componentService.newComponentAsync(
        this,
        function(newButton){
            //Pass an event handler to the new button
            newButton.addHandler('press', cmp, 'c.someHandler');

            //Add the new button to the body array
            var body = cmp.get("v.body");
            body.push(newButton);
            cmp.set("v.body", body);
        },
        {
            "componentDef": "markup://ui:button",
            "localId": "myLocalId",
            "attributes": {
                "values": { label: "Submit" }
            }
        }
    );
}
```

`$A.componentService.newComponentAsync()` is equivalent to `$A.newCmpAsync()`. To retrieve the new button you created, use `body[0]`.

```
var newbody = cmp.get("v.body");
var newCmp = body[0].find("myLocalId");
```

Note: The `componentDef` attribute represents the component definition you're creating. It contains the definition descriptor of the component in the format `markup://namespace:name`, which is a reference to the metadata of a component definition.

Declaring Dependencies

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 is not directly referenced in the component's markup. To tell the framework about such a dynamic dependency, use the `<aura:dependency>` tag. This ensures that the component and its dependencies are sent to the client, when needed.

For more information about usage, see aura:dependency on page 135.

Server-Side Dependencies

The `newComponentAsync()` method supports both client-side and server-side component creation. If no server-side dependencies are found, this method is run synchronously. The top-level component determines whether a server request is necessary for component creation.

Note: Creating components where the top-level components don't have server dependencies but nested inner components do is not currently supported.

A server-side controller is not a server-side dependency for component creation as controller actions are only called after the component has been created.

A component with server-side dependencies is created on the server, even if it's preloaded. If there are no server dependencies and the definition already exists on the client via preloading or declared dependencies, no server call is made. To force a server request, set the `forceServer` parameter to `true`.

SEE ALSO:

aura:component

Dynamically Adding Event Handlers

Dynamically Adding Event Handlers

You can dynamically add a handler for an event that a component fires. The component can be created dynamically on the client-side or fetched from the server at runtime.

This sample code adds an event handler to instances of `auradocs:sampleComponent`.

```
addNewHandler : function(cmp, event) {
    var cmpArr = cmp.find({ instancesOf : "auradocs:sampleComponent" });
    for (var i = 0; i < cmpArr.length; i++) {
        var outputCmpArr = cmpArr[i];
        outputCmpArr.addHandler("someAction", cmp, "c.someAction");
    }
}
```

You can also add an event handler to a component that is created dynamically in the callback function of `$A.services.component.newComponentAsync()`. See Dynamically Creating Components for more information.

`component.addHandler()` adds an event handler to a component. Note that you cannot force a component to start firing events that it doesn't fire. `c.someAction` can be an action in a controller in the component's hierarchy. `someAction` and `cmp` refers to the event name and value provider respectively. `someAction` must match the `name` attribute value in the `aura:registerEvent` or `aura:handler` tag. Refer to the JavaScript API reference for a full list of methods and arguments.

SEE ALSO:

- Handling Events with Client-Side Controllers
- Creating Server-Side Logic with Controllers
- Client-Side Rendering to the DOM

Modifying Components from External JavaScript

You can modify component state outside an event handler and trigger re-rendering of the component. This is particularly useful if you use `window.setTimeout()` in your event handlers to execute some logic after a time delay.

```
window.setTimeout(function () {
    $A.run(function() {
        cmp.set("v.visible", true);
    });
}, 5000);
```

This code sets the `visible` attribute on a component to `true` after a five-second delay. Use `$A.run()` to modify a component outside an event handler and trigger re-rendering of the component by the framework.

SEE ALSO:

- Handling Events with Client-Side Controllers
- Firing Lightning Events from Non-Lightning Code
- Events

Dynamically Showing or Hiding Markup

You can show or hide markup when a button is pressed.

Component source

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

    <br/>

    <p>Click the button to see toggling at its best!</p>

    <br/>

    <aura:renderIf isTrue="{!v.visible}">
        <p>Now, you see me!</p>
    </aura:renderIf>

    <ui:button label="Toggle Markup Visibility" press="{!c.showHide}"/>
```

```
</aura:component>
```

Client-side controller source

```
({
    showHide: function(cmp) {
        var isVisible = cmp.get("v.visible");
        // toggle the visible value
        cmp.set("v.visible", !isVisible);
    }
})
```

Let's look at the **Component source** to see how this works. We added an attribute called `visible` to control whether the markup is visible. It's set to `false` by default so that the markup is not visible. Under the covers, there are no DOM elements created for the markup.

The `aura:renderIf` tag selectively display the markup in its body if the `visible` attribute evaluates to `true`.

The `ui:button` triggers the `showHide` action in the client-side controller. It simply toggles the value of the `visible` attribute.

SEE ALSO:

- Handling Events with Client-Side Controllers
- Component Attributes
- aura:renderIf

Adding and Removing Styles

You can add or remove a CSS style to an element during runtime.

The following demo shows how to append and remove a CSS style from an element.

Component source

```
<aura:component>
    <div aura:id="changeIt">Change Me!</div><br />
    <ui:button press="{!c.applyCSS}" label="Add Style" />
    <ui:button press="{!c.removeCSS}" label="Remove Style" />

</aura:component>
```

CSS source

```
.THIS.changeMe {
    background-color:yellow;
    width:200px;
}
```

Client-side controller source

```
{
    applyCSS: function(cmp, event) {
        var el = cmp.find('changeIt');
        $A.util.addClass(el.getElement(), 'changeMe');
    },
```

```
    removeCSS: function(cmp, event) {
        var el = cmp.find('changeIt');
        $A.util.removeClass(el.getElement(), 'changeMe');
    }
}
```

The buttons in this demo are wired to controller actions that append or remove the CSS styles. To append a CSS style to an element, use `$A.util.addClass(element, 'class');`. Similarly, remove the class by using `$A.util.removeClass(element, 'class');` in your controller. `cmp.find()` locates the element using the local ID, denoted by `aura:id="changeIt"` in this demo.

To toggle the class, use `$A.util.toggleClass(element, 'class');`, which adds or removes the class depending on the presence of the class in the element. Refer to the JavaScript API Reference for more utility functions for working with DOM elements.

SEE ALSO:

- Handling Events with Client-Side Controllers
- CSS in Components
- Component Bundles

CHAPTER 11 Using Apex

In this chapter ...

Use Apex to write server-side code, such as controllers and test classes.

Server-side controllers handle requests from client-side controllers. For example, a client-side controller might handle an event and call a server-side controller action to persist a record. A server-side controller can also load your record data.

Working with Components

Apex uses a simple dot notation to work with components and component attributes.

To reference a component, use `Cmp.<myNamespace>.<myComponent>`; for example, `Cmp.ui.button`.

To reference a component's attribute, use `Cmp.<myNamespace>.<myComponent>.<myAttribute>`; for example, `Cmp.ui.button.label`.

Creating a Component in Apex

To create a component, use this syntax:

```
Cmp.<myNamespace>.<myComponent> cmpVar = new Cmp.<myNamespace>.<myComponent>();
```

For example:

```
Cmp.ui.button button = new Cmp.ui.button();
```

You can also include attributes when you're creating a component. For example:

```
Cmp.ui.button button = new Cmp.ui.button(label = 'Click Me');
```

Updating a Component Attribute

You can update an attribute value in a component by assigning a new value. For example:

```
Cmp.ui.button button = new Cmp.ui.button(label = 'Click Me');
String buttonLabel = button.label;
button.label = 'Click Me Not';
```

Accessing the Current Component

Use `Aura.getComponent()` to access the current component in the component's controller. For example, in a button's controller, you could access the button component like this.

```
Cmp.ui.button button = Aura.getComponent();
```

Working with Salesforce Records

It's easy to work with your Salesforce records in Apex.

The term `sObject` refers to any object that can be stored in Force.com. This could be a standard object, such as Account, or a custom object that you create, such as a Merchandise object.

An `sObject` variable represents a row of data, also known as a record. To work with an object in Apex, declare it using the SOAP API name of the object. For example:

```
Account a = new Account();
MyCustomObject__c co = new MyCustomObject__c();
```

For more information on working on records with Apex, see Working with Data in Apex.

This example controller persists an updated Account record. Note that the `update` method has the `@AuraEnabled` annotation, which enables it to be called as a server-side controller action.

```
public class AccountController {

    @AuraEnabled
    public static void updateAnnualRevenue(String accountId, Decimal annualRevenue) {
        Account acct = [SELECT Id, Name, BillingCity FROM Account WHERE Id = :accountId];

        acct.AnnualRevenue = annualRevenue;
        update acct;
    }
}
```

For an example of calling Apex code from JavaScript code, see the Quick Start on page 6.

Loading Record Data from A Custom Object

Load record data using an Apex server-side controller and setting the data on a component attribute. This server-side controller returns records on a custom object `myObj__c`.

```
public class MyObjController {

    @AuraEnabled
    public static List<MyObj__c> getMyObjects() {
        return [SELECT id, name, myField__c FROM MyObj__c];
    }
}
```

This example component uses the previous controller to display a list of records from the `myObj__c` custom object.

```
<aura:component controller="namespace.MyObjController"/>
<aura:attribute name="myObjects" type="namespace.MyObj__c[]"/>
<aura:iteration items="{!v.myObjects}" var="obj">
    {!obj.name}, {!obj.namespace__myField__c}
</aura:iteration>
```

This client-side controller sets the `myObjects` component attribute with the record data by calling the `getMyObjects()` method in the server-side controller.

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

For an example on loading and updating records using controllers, see the Quick Start on page 6.

Loading Record Data from a Standard Object

Similarly, you can load records from a standard object. This server-side controller has methods to return a list of opportunity records and an individual opportunity record.

```
public class OpportunityController {

    @AuraEnabled
    public static List<Opportunity> getOpportunities() {
        List<Opportunity> opportunities =
                [SELECT Id, Name, CloseDate FROM Opportunity];
        return opportunities;
    }

    @AuraEnabled
    public static Opportunity getOpportunity(Id id) {
        Opportunity opportunity = [
                SELECT Id, Account.Name, Name, CloseDate,
                       Owner.Name, Amount, Description, StageName
            FROM Opportunity
            WHERE Id = :id
         ];
        return opportunity;
    }
}
```

This example component uses the previous server-side controller to display a list of opportunity records.

```
<aura:component controller="namespace.OpportunityController">
    <aura:attribute name="opportunities" type="Opportunity[]"/>
    <aura:iteration var="opportunity" items="{!v.opportunities}">
                {!opportunity.Id} : {!opportunity.Name}
    </aura:iteration>
</aura:component>
```

To set the record data on a component attribute, call the `getOpportunities()` server-side controller from a client-side controller and set the `opportunities` attribute, as shown in the previous example. For more information about calling server-side controller methods, see Calling a Server-Side Action on page 119.

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 a record.

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.

For more details on the process of calling a server-side action, see Calling a Server-Side Action on page 119.

IN THIS SECTION:

Apex Server-Side Controller Overview

Create a server-side controller in Apex and use the `@AuraEnabled` annotation to enable client- and server-side access to the controller method.

All methods on server-side controllers must be static because the framework doesn't create a controller instance per component instance. Instead, all instances of a given component share one static controller.

Warning: Any state stored on the controller is shared across all instances of a component definition.

Only methods that you have explicitly annotated with `@AuraEnabled` are exposed. Other methods are not available.

This Apex controller contains a `serverEcho` action that prepends a string to the value passed in.

```
public class SimpleServerSideController {

    //Use @AuraEnabled to enable client- and server-side access to the method
    @AuraEnabled
    public static String serverEcho(String firstName) {
        return ('Hello from the server, ' + firstName);
    }
}
```

Creating an Apex Server-Side Controller

Use the Developer Console to create an Apex server-side controller.

You must use a Developer Edition organization with a registered namespace.

1. Click *Your name* > **Developer Console**.
2. Click **File** > **New** > **Apex Class**.
3. Enter a name for your server-side controller.
4. Click **OK**.

5. Enter a method for each server-side action in the body of the class.
6. Click **File** > **Save**.
7. Open the component that you want to wire to the new controller class.
8. Add a `controller` system attribute to the `<aura:component>` tag to wire the component to the controller. For example:

```
<aura:component controller="myNamespace.MyApexController" >
```

Calling a Server-Side Action

Call a server-side controller action from a client-side controller. In the client-side controller, you set a callback, which is called after the server-side action is completed. A server-side action can return any object containing serializable JSON data.

A client-side controller is a JSON object containing name-value pairs. Each name corresponds to a client-side action. Its value is the JavaScript function associated with the action.

The following client-side controller includes an `echo` action that executes a `serverEcho` action on a server-side controller. The client-side controller sets a callback action that is invoked after the server-side action returns. In this case, the callback function alerts the user with the value returned from the server.

```
{
    "echo" : function(component) {
        // create a one-time use instance of the serverEcho action
        // in the server-side controller
        var a = component.get("c.serverEcho");
        a.setParams({ firstName : component.get("v.firstName") });

        // Create a callback that is executed after
        // the server-side action returns
        a.setCallback(this, function(action) {
            if (action.getState() === "SUCCESS") {
                // Alert the user with the value returned
                // from the server
                alert("From server: " + action.getReturnValue());

                // You would typically fire a event here to trigger
                // client-side notification that the server-side
                // action is complete
            }
            else if (action.getState() === "ERROR"){
                var errors = a.getError();
                if (errors) {
                    $A.logf("Errors", errors);
                    if (errors[0] && errors[0].message) {
                        $A.error("Error message: " +
                                 errors[0].message);
                    }
                } else {
                    $A.error("Unknown error");
                }
            }
            else {
                alert("Action state: " + action.getState());
            }
```

```
        });

        // A client-side action could cause multiple events,
        // which could trigger other events and
        // other server-side action calls.
        // $A.enqueueAction adds the server-side action to the queue.
        $A.enqueueAction(a);
    }
}
```

In the client-side controller, we use the value provider of `c` to invoke a server-side controller action. This is the same syntax as we use in markup to invoke a client-side controller action. The `cmp.get("c.serverEcho")` call indicates that we are calling the `serverEcho` method in the server-side controller. The method name in the server-side controller must match everything after the `c.` in the client-side call.

Use `$A.enqueueAction(action)` to add the server-side controller action to the queue of actions to be executed. All actions that are enqueued this way will be run at the end of the event loop. Rather than sending a separate request for each individual action, the framework processes the event chain and executes the action in the queue after batching up related requests. The actions are asynchronous and have callbacks.

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

ABORTED
The action was aborted

SEE ALSO:

Handling Events with Client-Side Controllers

Queueing of Server-Side Actions

Queueing of Server-Side Actions

The framework queues up actions before sending them to the server. This mechanism is largely transparent to you when you're writing code but it enables the framework to minimize network traffic.

Event processing can generate a tree of events if an event handler fires more events. The framework processes the event tree and adds every action that needs to be executed on the server to a queue.

When the tree of events and all the client-side actions are processed, the framework batches actions from the queue into a message before sending it to the server. A message is essentially a wrapper around a list of actions.

Abortable Actions

You can mark an action as abortable to make it potentially abortable while it's queued to be sent to the server or not yet returned from the server. This is useful for actions that you'd like to abort when there is a newer abortable action in the queue.

A set of actions for a single transaction, such as a click callback, are queued together to be sent to the server. If a user starts another transaction, for example by clicking another button, all abortable actions are removed from the queue. The aborted actions are not sent to the server and their state is set to `ABORTED`. If some actions have not yet returned from the server, they will complete, but their callbacks will not be called. An abortable action is sent to the server and executed normally unless it hasn't returned from the server when a subsequent abortable action is added to the queue.

Note: There is no requirement that the most recent abortable action has to be identical to the previous abortable actions. The most recent action just has to be marked as abortable.

Mark a server-side action as abortable by using the `setAbortable()` method on the `Action` object in JavaScript. For example:

```
var a = component.get("c.serverEcho");
a.setAbortable();
```

You can check for aborted actions in your callback and take appropriate action, such as logging the aborted action, if desired. For example:

```
a.setCallback(this, function(action) {
    if (action.getState() === "SUCCESS") {
        // Alert the user with the value returned from the server
        alert("From server: " + action.getReturnValue());
    }
    else if (action.getState() === "ABORTED") {
        alert("The action was aborted");
    }
    else { // something bad happened
        alert("Action state: " + action.getState());
    }
});
```

SEE ALSO:

- Creating Server-Side Logic with Controllers
- Queueing of Server-Side Actions
- Calling a Server-Side Action

Testing Your Apex Code

Before you can upload a managed package, you must write and execute tests for your Apex code to meet minimum code coverage requirements. Also, all tests must run without errors when you upload your package to AppExchange.

To package your application and components that depend on Apex code, the following must be true.

- At least 75% of your Apex code must be covered by unit tests, and all of those tests must complete successfully.

  Note the following.

  - When deploying to a production organization, every unit test in your organization namespace is executed.
  - Calls to `System.debug` are not counted as part of Apex code coverage.
  - Test methods and test classes are not counted as part of Apex code coverage.

  - While only 75% of your Apex code must be covered by tests, your focus shouldn't be on the percentage of code that is covered. Instead, you should make sure that every use case of your application is covered, including positive and negative cases, as well as bulk and single records. This should lead to 75% or more of your code being covered by unit tests.

- Every trigger must have some test coverage.
- All classes and triggers must compile successfully.

This sample shows an Apex test class that is used with the controller class in the expense tracker app available at Create A Standalone Lightning App on page 10.

```
@isTest
class TestExpenseController {
    static testMethod void test() {
        //Create new expense and insert it into the database
        Expense__c exp = new Expense__c(name='My New Expense',
                                amount__c=20, client__c='ABC',
                                reimbursed__c=false, date__c=null);
        insert exp;

        ExpenseController c = new ExpenseController();
        //Assert the name field and saved expense
        System.assertEquals('My New Expense',
                           ExpenseController.getExpenses()[0].Name,
                           'Name does not match');
        System.assertEquals(exp, ExpenseController.saveExpense(exp));
    }
}
```

Note: Apex classes must be manually added to your package.

For more information on distributing Apex code, see the *Apex Code Developer's Guide*.

SEE ALSO:

Distributing Applications and Components

CHAPTER 12 Using Interfaces

In this chapter ...

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, the Lightning Component framework 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 `<aura:attribute>` tags that define the interface's attributes. You can't use markup or controllers 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" >
```

Note: To set the value of an attribute inherited from an interface, you must redefine the attribute in the sub-component using `<aura:attribute>` and set the value in its default attribute.

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")`.

CHAPTER 13 Using the AppCache

In this chapter ...

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.

SEE ALSO:

aura:application

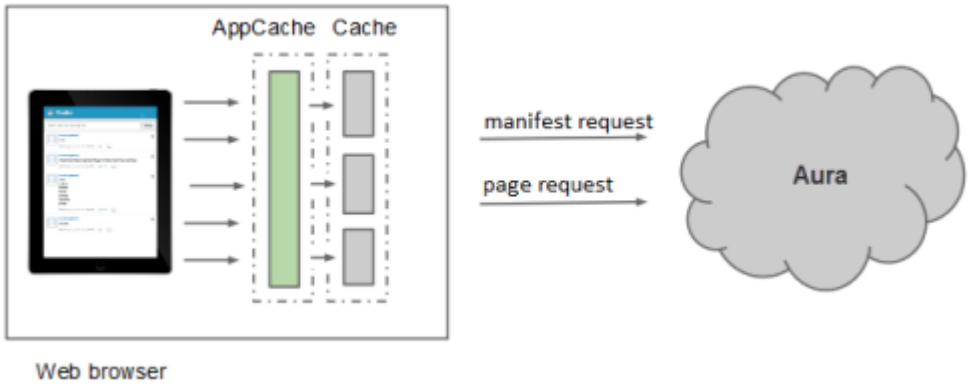
Enabling the AppCache

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.

When a browser initially requests an app, a link to the manifest file is included in the response. The browser retrieves the resource files that are listed in the manifest file, such as the JavaScript and CSS files, and they are cached in the browser cache. Finally, the browser fetches a copy of the manifest file and downloads all resources listed in the manifest file and stores them in the AppCache.

CHAPTER 14 Controlling Access

In this chapter ...

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

Tag	Description
`aura:application`	Represents an application
`aura:interface`	Represents an interface
`aura:component`	Represents a component
`aura:attribute`	Represents an attribute in an application, interface, component, or event
`aura:event`	Represents an event

By default, the `access` attribute is set to `public` for all tags, which allows them to be extended or used within the same namespace.

Application Access Control

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

Possible values are listed below.

Modifier	Description
`global`	The app can be extended by another app in any namespace if `extensible="true"` is set on the `aura:application` tag.
`public`	The app can be extended by another app within the same namespace only. This is the default access level.

Interface Access Control

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

Possible values are listed below.

Modifier	Description
`global`	The interface can be extended by another interface or used by a component in any namespace.
`public`	The interface can be extended by another interface or used by a component within the same namespace only. This is the default access level.

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

Component Access Control

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

Possible values are listed below.

Modifier	Description
`global`	The component can be used by another component or application in any namespace. It can also be extended in any namespace if `extensible="true"` is set on the `aura:component` tag.
`public`	The component can be extended or used by another component, or used by an application within the same namespace only. This is the default access level.

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

Attribute Access Control

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

Possible values are listed below.

Access	Description
`global`	The attribute can be used in any namespace.
`public`	The attribute can be used within the same namespace only. This is the default access level.
`private`	The attribute can be used only within the container app, interface, component, or event, and can't be referenced externally.

Event Access Control

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

Possible values are listed below.

Modifier	Description
`global`	The event can be used or extended in any namespace.
`public`	The event can be used or extended within the same namespace only. This is the default access level.

CHAPTER 15 Distributing Applications and Components

As an ISV or Salesforce partner, you can package and distribute applications and components to other Salesforce users and organizations, including those outside your company.

Publish applications and components to and install them from AppExchange. When adding an application or component to a package, all definition bundles referenced by the application or component are automatically included, such as other components, events, and interfaces. Custom fields, custom objects, list views, page layouts, and Apex classes referenced by the application or component are also included. However, when you add a custom object to a package, the application and other definition bundles that reference that custom object must be explicitly added to the package.

To create and work with managed or unmanaged packages, you must use a Developer Edition organization and register a namespace prefix. A namespace prefix is required for creating applications and components in the Developer Console. To ensure that your application and other resources are fully upgradeable, use a managed package. A managed package includes your namespace prefix in the component names and prevents naming conflicts in an installer's organization. An organization can create a single managed package that can be downloaded and installed by other organizations. After installation from a managed package, the application or component names are locked, but the following attributes are editable.

- API Version
- Description
- Label
- Language
- Markup

Any Apex that is included as part of your definition bundle must have at least 75% cumulative test coverage. When you upload your package to AppExchange, all tests are run to ensure that they run without errors. The tests are also run when the package is installed.

For more information on packaging and distributing, see the *ISVforce Guide*.

SEE ALSO:

Testing Your Apex Code

CHAPTER 16 Debugging

In this chapter ...

There are a few basic tools that can help you to debug applications.

For example, use Chrome Developer Tools to debug your client-side code.

- To open Developer Tools 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 Chrome Developer Tools website.

Debugging JavaScript Code

Enable debug mode to make it easier to debug JavaScript code in your Lightning components.

By default, the Lightning Component framework runs in `PROD` mode. This mode is optimized for performance. It uses the Google Closure Compiler to optimize and minimize the size of the JavaScript code. The method name and code are heavily obfuscated.

When you enable debug mode, the framework runs in `PRODDEBUG` mode by default. It doesn't use Google Closure Compiler so the JavaScript code isn't minimized and is easier to read and debug.

To enable debug mode:

1. From Setup, click **Develop** > **Lightning Components**.
2. Select the `Enable Debug Mode` checkbox.
3. Click **Save**.

EDITIONS

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

Log Messages

To help debug your client-side code, you can use the `log()` method to write output to the JavaScript console of your web browser.

Use the `$A.log(string, [error])` method to output a log message to the JavaScript console. The first parameter is the string to log and the optional second parameter is an error object whose messages should be logged. For example, `$A.log("This is a log message");` will output "This is a log message" to the JavaScript console. If you put `$A.log("The name of the action is: " + this.getDef().getName());` inside an action called "openNote" in a client-side controller, then the log message "The name of the action is: openNote" will be output to the JavaScript console.

For instructions on using the JavaScript console, refer to the instructions for your web browser.

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.");` will output "This is a warning message." to the JavaScript console. A stack trace will also be displayed in the JavaScript console.

For instructions on using the JavaScript console, refer to the instructions for your web browser.

CHAPTER 17 Reference Overview

In this chapter ...

This section contains reference documentation including details of the various tags available in the framework.

Reference Doc App

The reference doc app includes more reference information, including descriptions and source for the out-of-the-box components that come with the framework. Access the app at:

`https://<mySalesforceInstance>.lightning.force.com/auradocs/reference.app`, where `<mySalesforceInstance>` is the name of the instance hosting your org; for example, `na1`.

aura:application

An app is a special top-level component whose markup is in a `.app` resource.

The markup looks similar to HTML and can contain components as well as a set of supported HTML tags. The `.app` resource is a standalone entry point for the app and enables you to define the overall application layout, style sheets, and global JavaScript includes. It starts with the top-level `<aura:application>` tag, which contains optional system attributes. These system attributes tell the framework how to configure the app.

System Attribute	Type	Description
`access`	String	Indicates whether the app can be extended by another app outside of a namespace. Possible values are `public` (default), and `global`.
`controller`	String	The server-side controller class for the app. The format is `namespace.myController`.
`description`	String	A brief description of the app.
`implements`	String	A comma-separated list of interfaces that the app implements.
`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:

- App Basics
- Using the AppCache
- Application Access Control

aura:component

A component is represented by the `aura:component` tag, which has the following optional attributes.

Attribute	Type	Description
access	String	Indicates whether the component can be used outside of its own namespace. Possible values are public (default), and global.
controller	String	The server-side controller class for the component. The format is namespace.myController.
description	String	A description of the component.
implements	String	A comma-separated list of interfaces that the component implements.

aura:component also includes a body attribute defined in a <aura:attribute> tag. Attributes usually control the output or behavior of a component, but not the configuration information in system attributes.

Attribute	Type	Description
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.

SEE ALSO:

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

aura: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;

System Attribute	Description
	`resource="markup://sampleNamespace:input*"` matches everything in the namespace that starts with `input`.
`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:

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 `public` (default), and `global`.
`description`	String	A description of the event.
`extends`	Component	The event to be extended. For example, `extends="namespace:myEvent"`.
`type`	String	Required. Possible values are `COMPONENT` or `APPLICATION`.

SEE ALSO:

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 on the server and instantiates components either in its `body` or `else` attribute.

Note: `aura:if` instantiates the components in either its body or the `else` attribute, but not both. `aura:renderIf` instantiates both the components in its body and the `else` attribute, but only renders one. If the state of `isTrue` changes, `aura:if` has to first instantiate the components for the other state and then render them. We recommend using `aura:if`

instead of `aura:renderIf` to improve performance. Only consider using `aura:renderIf` if you expect to show the components for both the `true` and `false` states, and it would require a server round trip to instantiate the components that aren't initially rendered. Otherwise, use `aura:if` to render content if a provided expression evaluates to true.

Attribute Name	Type	Description
`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.

SEE ALSO:

aura:renderIf

aura:interface

The `aura:interface` tag has the following optional attributes.

Attribute	Type	Description
`access`	String	Indicates whether the interface can be extended or used outside of its own namespace. Possible values are `public` (default), 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"`.

SEE ALSO:

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`.

Attribute Name	Type	Description
`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.
`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.

Example Using Data from a Server-Side Controller

This example shows a dynamic iteration that displays data from a custom object.

```
<aura:component controller="namespace.MyObjectController">
<aura:attribute name="myObject" type="namespace.myObject__c[]"/>
  <aura:iteration items="{!v.myObject}" var="obj">
    <!-- Display names of all accounts -->
    {!obj.name}, {!obj.namespace__myField__c}
  </aura:iteration>
</aura:component>
```

aura:renderIf

`aura:renderIf` renders the content within the tag if the `isTrue` attribute evaluates to `true`.

Only consider using `aura:renderIf` if you expect to show the components for both the `true` and `false` states, and it would require a server round trip to instantiate the components that aren't initially rendered. Otherwise, use `aura:if` to render content if a provided expression evaluates to true.

Attribute Name	Type	Description
`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.

Passing in an Expression

Use `aura:renderIf` if you are passing an expression into a component to be evaluated. For example, you have a container component that references a component, which has a `aura:renderIf` tag.

container.cmp

```
<aura:attribute name="native" type="Boolean" default="true"/>
<auradocs:myCmp value="0.5" native={!v.native || v.native}"/>
<ui:button label="Toggle" press="{!c.toggleMe}"/>
```

myCmp.cmp

```
<aura:attribute name="native" type="Boolean" default="true"/>
<aura:attribute name="value" type="Decimal"/>

<aura:renderIf isTrue="{!v.native}">
  <meter value="{!v.value}">{!(v.value * 100) + '%'}</meter>
  <aura:set attribute="else">
    <!--your meter here-->
  </aura:set>
</aura:renderIf>
```

The container component has a button which toggles the `native` attribute value.

containerController.js

```
({
    toggleMe: function(cmp) {
        cmp.set('v.native', !cmp.get('v.native'));
    }
})
```

When the button is pressed, the expression `native={!v.native || v.native}"` is passed into the `aura:renderIf` tag and reevaluated correctly.

SEE ALSO:

aura:if

aura:set

Use the `<aura:set>` system tag to set the value of an attribute on a component reference, or on an event or interface. When you include another component, such as `<ui:button>`, in a component, we call that a component reference to `<ui:button>`.

To learn more, see:

- Setting Attributes on a Component Reference
- Setting Attributes Inherited from an Interface

Setting Attributes on a Component Reference

When you include another component, such as `<ui:button>`, in a component, we call that a component reference to `<ui:button>`. You can use `<aura:set>` to set an attribute on the component reference. For example, if your component includes a reference to `<ui:button>`:

```
<ui:button label="">
    <aura:set attribute="label" value="hello"/>
</ui:button>
```

This is equivalent to:

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

The latter syntax without `aura:set` makes more sense in this simple example.

`aura:set` is more useful when you want to set markup as the attribute value. For example, the `<aura:set>` tag specifies the markup for the `else` attribute in the `aura:if` component.

```
<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">
           Show this if condition is false
        </aura:set>
    </aura:if>
</aura:component>
```

Setting Attributes Inherited from an Interface

To set the value of an attribute inherited from an interface, redefine the attribute in the component. For example, a component implements an interface that has an attribute `myBoolean` set to `false`. The following example sets `myBoolean` on the component to `true`.

```
<aura:component implements="auradocs:myIntf">
    <aura:attribute name="myBoolean" type="Boolean" default="true" />
</aura:component>
```

If the component that implements the interface is contained in another component, you can use `aura:set` as discussed. Alternatively, you can set the attribute value like this.

```
<aura:component>
    <auradocs:sampleCmp myBoolean="true" />
</aura:component>
```

Supported HTML Tags

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

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`
- `svg`

Supported aura:attribute Types

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

Attribute Name	Type	Description
`access`	String	Indicates whether the attribute can be used outside of its own namespace. Possible values are `public` (default), and `global`, and `private`.
`name`	String	Required. The name of the attribute. For example, if you set `<aura:attribute name="isTrue" type="Boolean" />` on a component called `aura:newCmp`, you can set this attribute when you 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. You can't use an expression to set the default value of an attribute. Instead, to set a dynamic default, use an `init` event. See Invoking Actions on Component Initialization.
`required`	Boolean	Determines if the attribute is required. The default is `false`.
`description`	String	A summary of the attribute and its usage.

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

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

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

SEE ALSO:

Component Attributes

Basic Types

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

type	Example	Description
`Boolean`	`<aura:attribute name="showDetail" type="Boolean" />`	Valid values are `true` or `false`. To set a default value of `true`, add `default="true"`.
`Date`	`<aura:attribute name="startDate" type="Date" />`	A date corresponding to a calendar day in the format yyyy-mm-dd. The hh:mm:ss portion of the date 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`.

type	Example	Description
`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[]" />
```

Retrieving Data from an Apex Controller

To retrieve the string array from an Apex controller, bind the component to the controller. This component retrieves the string array when a button is clicked.

```
<aura:component controller="namespace.AttributeTypes">
    <aura:attribute name="favoriteColors" type="String[]" default="cyan, yellow, magenta"/>

    <aura:iteration items="{!v.favoriteColors}" var="s">
        {!s}
    </aura:iteration>
    <ui:button press="{!c.getString}" label="Update"/>
</aura:component>
```

Set the Apex controller to return a `List<String>` object.

```
public class AttributeTypes {
    private final String[] arrayItems;

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

}
```

This client-side controller retrieves the string array from the Apex controller and displays it using the `{!v.favoriteColors}` expression.

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

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();
```

Standard and Custom Object Types

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

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

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

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

Collection Types

Here are the supported collection type values.

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="1,2,3" />`	A collection that contains no duplicate elements. The order for set items is not guaranteed. For example, `"1,2,3"` might be returned as `"3,2,1"`.

Setting List Items

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

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

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

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

To retrieve list data from a controller, you can use `aura:iteration`. See aura:iteration on page 137 for more information.

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
}
```

Custom Apex Class Types

An attribute can have a type corresponding to an Apex class. For example, this is an attribute for a `Color` Apex class:

```
<aura:attribute name="color" type="docSampleNamespace.Color" />
```

Support for Collections

If an attribute can contain more than one element, use a `List` or an array.

This `aura:attribute` tag shows the syntax for a `List` of Apex objects:

```
<aura:attribute name="colorPalette" type="List<docSampleNamespace.Color>" />
```

This `aura:attribute` tag shows the syntax for an array of Apex objects:

```
<aura:attribute name="colorPalette" type="docSampleNamespace.Color[]" />
```

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.

SEE ALSO:

- Component Body
- Component Facets

INDEX

D

E

F

G

H

I

J

L

M

N