Lightning Components Developer's Guide

Version 34.0, Summer '15

CONTENTS

Contents

Contents

Contents

Contents

Contents

Contents

GETTING STARTED

CHAPTER 1 Introduction

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 <mark>is a UI framework for developing</mark> 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 <mark>uses JavaScript on the client side and Apex on the server side.</mark>

> 📝 **Note:** Lightning components in standalone apps (`.app` resources) are generally available. Integrating Lightning components with Salesforce1 is in a beta release that is production quality but has some limitations.

Lightning Component Framework and Aura Framework

The Lightning Component framework is built on the open-source Aura framework available at http://github.com/forcedotcom/aura. You can build general-purpose web apps hosted on your own server using the Aura framework, which also provides features and components that may not be available in the Lightning Component framework. Many of the sample code here uses out-of-the-box components from the Aura framework, such as `aura:iteration` and `ui:button`. The `aura` namespace contains components to simplify your app logic, and the `ui` namespace contains components for user interface elements like buttons and input fields. The `force` namespace contains components specific to Salesforce.

Visualforce and Lightning

The Lightning Component framework is a component-based framework. As building blocks of an app, <mark>components encapsulate HTML, JavaScript, and CSS, while interacting via events.</mark> Lightning components are client-side centric, making them more dynamic and mobile friendly. Contrastingly, Visualforce components are page-centric and rely heavily on server calls. Visualforce facilitates delivering of template-driven web pages and email messages, making it suitable for developers who wish to maintain greater control over the lifecycle of the request. Because Lightning components use Apex on the server side, <mark>organizations that can't use Apex code can't create Lightning components, but they can use Visualforce.</mark>

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 client calls the server only when absolutely necessary; for example to get more metadata or data. The server only sends data that is needed by the user to maximize efficiency. The framework <mark>uses JSON to exchange data between the server and the client.</mark> It intelligently utilizes your server, browser, devices, and network so you can focus on the logic and interactions of your apps.

Event-driven architecture
Uses an <mark>event-driven architecture for better decoupling between components.</mark> Any component can <mark>subscribe to an application event, or to a component event</mark> 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 use responsive design 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®	
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**.
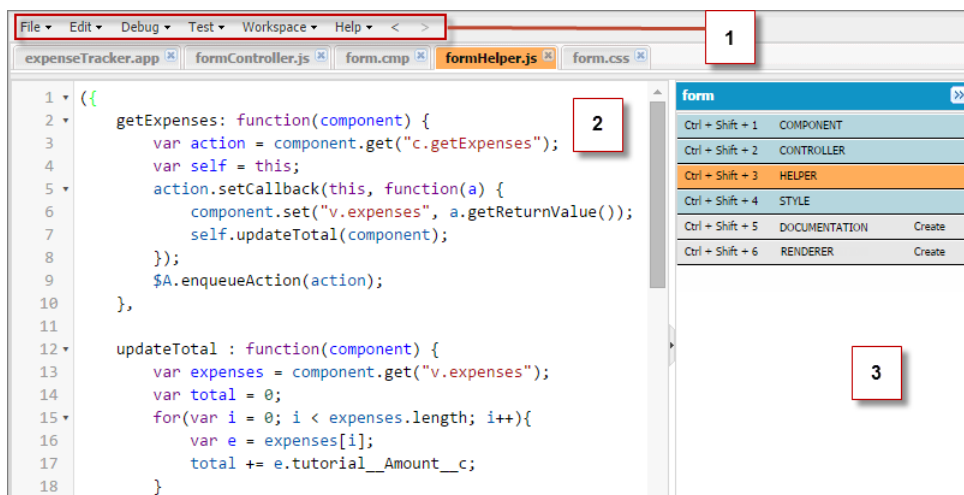
The Developer Console enables you to perform these functions.

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

  - Application

  - Component

  - Interface

  - Event

- 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.

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 Lightning apps and components , follow these prerequisites.

1. Create a Developer Edition organization
2. Register a Namespace Prefix
3. Enable Lightning Components in Salesforce1

> **Note:** You can create Lightning components using the UI in **Enterprise**, **Performance**, **Unlimited**, **Developer** Editions or a sandbox. If you don't plan to use a Developer Edition organization, you can go directly to Enable Lightning Components in Salesforce1. A namespace prefix is required if you plan to offer managed packages on the AppExchange. We're not using a namespace in this quick start tutorial.

Create a Developer Edition Organization

To register a namespace prefix, you need a Developer Edition organization, or *DE org* for short. If you don't have a DE org, create one 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.

> **Note:** A namespace is not required to create Lightning components but it's required if you plan to offer managed packages. If your org doesn't have a namespace, you can use the default namespace to access your components.

To register a namespace prefix:

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

   > **Note:** This button doesn't appear if you've already configured your developer settings.

3. Review the selections that are required for configuring developer settings, and then click **Continue**.
4. Enter the namespace prefix you want to register.
5. Click **Check Availability** to determine if the namespace prefix is already in use.
6. If the namespace prefix that you entered isn't available, repeat the previous two steps.
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 in Salesforce1

You must opt in to enable Lightning components in Salesforce1. Lightning components in standalone apps (`.app` resources) are available by default.

1. From Setup, click **Develop** > **Lightning Components**.

2. Select the `Enable Lightning Components` checkbox.

> ⚠ Warning: This beta version of Lightning components in Salesforce1 doesn't support Force.com Canvas apps in Salesforce1. Any Force.com Canvas apps in your organization will no longer work in Salesforce1 if you enable Lightning components.

3. Click **Save**.

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 11. 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 a component, you can include it in Salesforce1 by following the steps in Adding Lightning Components to Salesforce1 on page 58. For packaging and distributing your components and apps on AppExchange, see Distributing Applications and Components on page 180.

> 📝 Note: This app can be used in Salesforce1 and independent of Salesforce1 as a standalone app. To create components that utilize existing Salesforce1 form components and can be used only in Salesforce1, see Create a Component for Salesforce1 on page 30.

The following images show the expense tracker in Salesforce1 and as a standalone app.

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. The counter turns red when the sum exceeds $100.

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
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

Resources	Description
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
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

Optional: Install the Expense Tracker App

If you want to skip over the quick start tutorial, you can install the Expense Tracker app as a package.

A package is a bundle of components that you can install in your org. This packaged app is useful if you want to learn about the Lightning app without going through the quick start tutorial. If you're new to Lightning components, we recommend that you go through the quick start tutorial.

Note: Install the package in an org that doesn't have any of the objects with the same API name as the quick start objects.

To install the Expense Tracker app:

1. Click the installation URL link: https://login.salesforce.com/packaging/installPackage.apexp?p0=04to00000003onL
2. Log in to your organization by entering your username and password.
3. On the Package Installation Details page, click **Continue**.
4. Click **Next**, and on the Security Level page click **Next**.
5. Click **Install**.
6. Click **Deploy Now** and then **Deploy**.

When the installation completes, you can select the **Expenses** tab on the user interface to add new expense records.
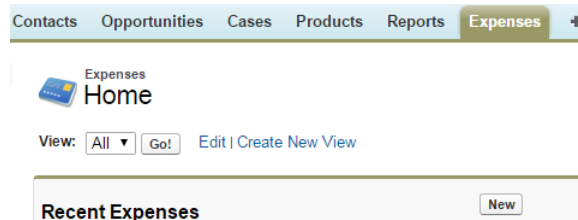
You'll also see the Expenses menu item on the Salesforce1 navigation menu. If you don't see the menu item in Salesforce1, add it by going to **Mobile Administration** > **Mobile Navigation**.
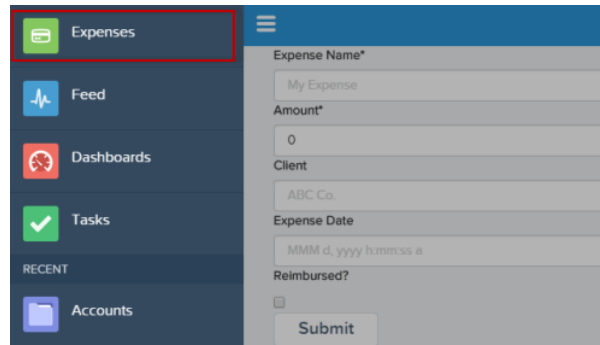
Next, you can modify the code in the Developer Console or explore the standalone app at
`https://<mySalesforceInstance>.lightning.force.com/c/expenseTracker.app`, where
`<mySalesforceInstance>` is the name of the instance hosting your org; for example, `na1`.

📝 Note: To delete the package, click **Setup** > **Installed Package**, and delete the package.

Create an Expense Object

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

You'll need to create this object if you're following the tutorial at Create a Standalone Lightning App on page 8.

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.

    📝 Note: If you're using a namespace prefix, you might see `namespace__Expense__c` instead of `Expense__c`.

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.

Custom Object
Expense

Help for this Page ⑦

Custom Object Definition Detail

[Edit] [Delete]

Singular Label	Expense	Description	
Plural Label	Expenses	Enable Reports	☐
Object Name	Expense	Track Activities	☐
API Name	Expense__c	Allow Sharing	✓
		Allow Bulk API Access	✓
		Allow Streaming API Access	✓
		Track Field History	☐
		Deployment Status	Deployed
		Help Settings	Standard salesforce.com Help Window
Created By	Jane Smith, 3/26/2015 11:04 AM	Modified By	Jane Smith, 3/26/2015 11:04 AM

Standard Fields

Standard Fields Help ⑦

Action	Field Label	Field Name	Data Type	Controlling Field	Indexed
	Created By	CreatedBy	Lookup(User)		
Edit	Expense Name	Name	Text(80)		✓
	Last Modified By	LastModifiedBy	Lookup(User)		
Edit	Owner	Owner	Lookup(User,Queue)		✓

Custom Fields & Relationships

[New] [Field Dependencies]

Custom Fields & Relationships Help ⑦

Action	Field Label	API Name	Data Type	Indexed	Controlling Field	Modified By
Edit	Del	Amount	Amount__c	Number(16, 2)			Jane Smith, 3/26/2015 11:04 AM
Edit	Del	Client	Client__c	Text(20)			Jane Smith, 3/26/2015 11:04 AM
Edit	Del	Date	Date__c	Date/Time			Jane Smith, 3/26/2015 11:04 AM
Edit	Del	Reimbursed?	Reimbursed__c	Checkbox			Jane Smith, 3/26/2015 11:04 AM

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	21		4/1/2015 12:00 PM	Unchecked
Dinner	70	ABC Co.	3/30/2015 7:00 PM	Checked

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.
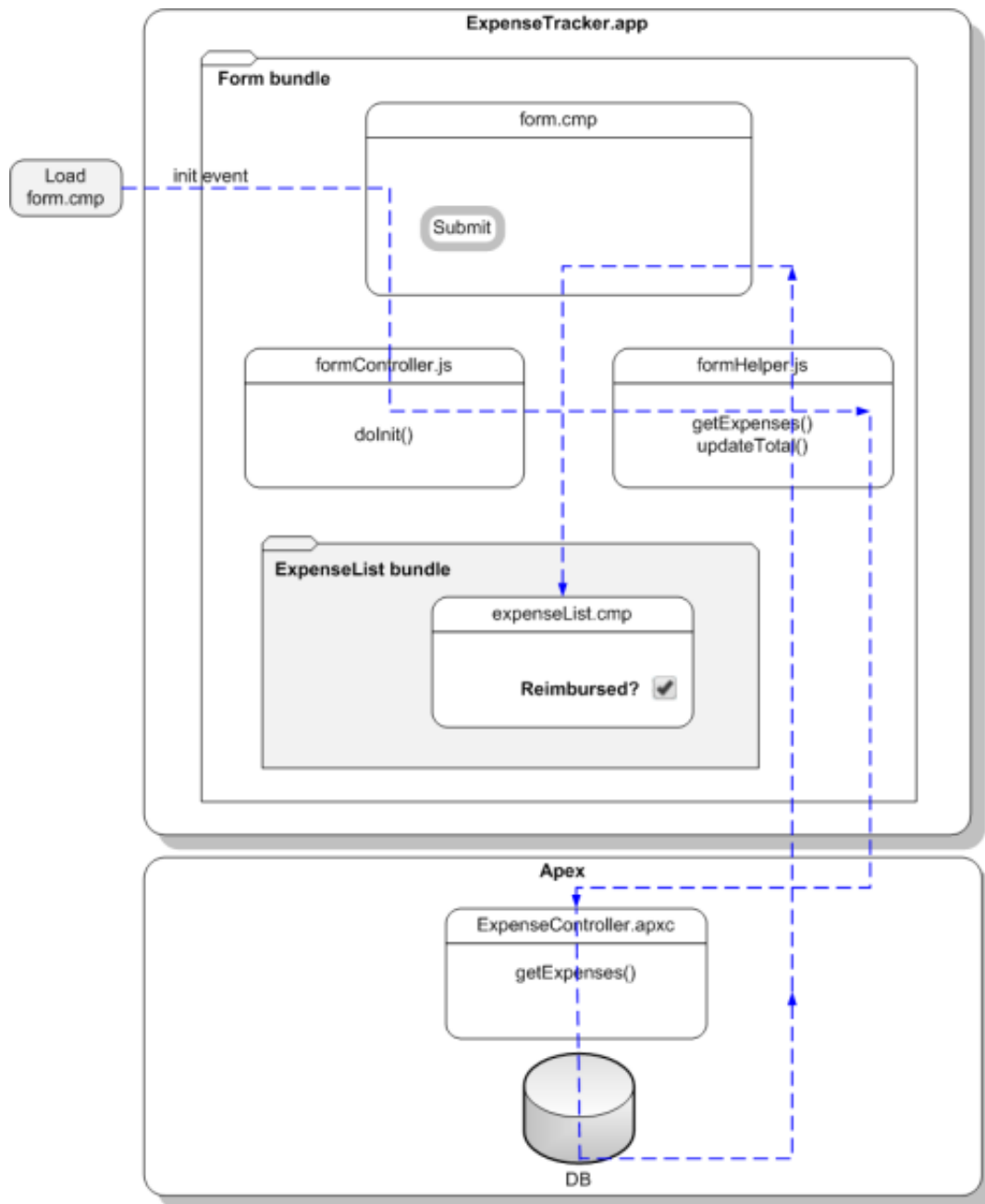
1. Click **Your Name** > **Developer Console** to open the Developer Console.

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>
    <ltng:require styles="/resource/bootstrap"/>
    <div class="bootstrap-sf1">
        <div class="navbar navbar-inverse">
```

```
        <div class="navbar-header">
            <a href="#" class="navbar-brand">My Expenses</a>
        </div>
        </div>
    </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 next. 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. Download the Salesforce1 styled theme available at http://developer.salesforcefoundation.org/bootstrap-sf1/ and upload the `bootstrap-namespaced.min.css` file as a static resource. This file is located in the `/dist/css` directory. This CSS resource includes the `.bootstrap-sf1` selector, which prevents styling conflict with other namespaces.

    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-namespaced.min.css` file. We'll not be uploading the other files to simplify this tutorial.

    c. Click **Save**.

6. 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`. If you're not using a namespace, your app is available at `/c/expenseTracker.app`.

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.

    > **Note:** The following code creates an input form that takes in user input to create an expense, which works in both a standalone app and in Salesforce1. For Lightning apps specific to Salesforce1, you can use `force:createRecord` to open the create record page.

```
<aura:component>
  <ltng:require styles="/resource/bootstrap"/>
  <aura:attribute name="expenses" type="Expense__c[]"/>
  <aura:attribute name="newExpense" type="Expense__c"
          default="{ 'sobjectType': 'Expense__c',
                     'Name': '',
                     'Amount__c': 0,
                     'Client__c': '',
```

```
                                    'Date__c': '',
                                    'Reimbursed__c': false
                                }"/>
    <!-- If you registered a namespace, replace line 3-11 with the following -->
    <!-- <aura:attribute name="expenses" type="myNamespace.Expense__c[]"/> -->
    <aura:attribute name="newExpense" type="myNamespace__Expense__c"
                default="{ 'sobjectType': 'myNamespace__Expense__c',
                            'Name': '',
                            'myNamespace__Amount__c': 0,
                            'myNamespace__Client__c': '',
                            'myNamespace__Date__c': '',
                            'myNamespace__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 -->
    <div class="bootstrap-sf1">
        <div class="container">
            <form>
                <fieldset>
                    <ui:inputText aura:id="expname" label="Expense Name"
                                    class="form-control"
                                    value="{!v.newExpense.Name}"
                                    placeholder="My Expense" required="true"/>

                <!-- If you registered a namespace, the attribute values must include
 your namespace -->
                    <!-- For example, value="{!v.newExpense.myNamespace__Amount__c}"
-->
                    <ui:inputNumber aura:id="amount" label="Amount"
                                    class="form-control"
                                    value="{!v.newExpense.Amount__c}"
                                    placeholder="20.80" required="true"/>
                    <ui:inputText aura:id="client" label="Client"
                                    class="form-control"
                                    value="{!v.newExpense.Client__c}"
                                    placeholder="ABC Co."/>
                    <ui:inputDateTime aura:id="expdate" label="Expense Date"
                                    class="form-control"
                                    value="{!v.newExpense.Date__c}"
                                    displayDatePicker="true"/>
                    <ui:inputCheckbox aura:id="reimbursed" label="Reimbursed?"
                                    class="checkbox"
                                    value="{!v.newExpense.Reimbursed__c}"/>
                    <ui:button label="Submit" press="{!c.createExpense}"/>
                </fieldset>
            </form>
    </div><!-- ./container-->

    <!-- Expense Counters -->
    <div class="container">
        <div class="row">
```

```
            <div class="col-sm-6">
              <!-- Make the counter red if total amount is more than 100 -->
              <div class="{!v.total >= 100 ? 'alert alert-danger' : 'alert
alert-info'}">
                    <h3>Total Expenses</h3>$<ui:outputNumber value="{!v.total}"
format=".00"/>
              </div>
            </div>
            <div class="col-sm-6">
                <div class="alert alert-info">
                    <h3>No. of Expenses</h3><ui:outputNumber value="{!v.exp}"/>
                </div>
             </div>
        </div>
  </div>
        <!-- Display expense records -->
        <div class="container">
            <div id="list" class="row">
              <aura:iteration items="{!v.expenses}" var="expense">
                    <!-- If you're using a namespace, use the format
{!expense.myNamespace__myField__c} instead. -->
              <p>{!expense.Name}, {!expense.Client__c}, {!expense.Amount__c},
{!expense.Date__c}, {!expense.Reimbursed__c}</p>
                </aura:iteration>
            </div>
        </div>
    </div><!--./bootstrap-sf1-->
</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 287.

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.Client__c}` represents data binding to the client field in the expense object.

> 📝 **Note:** The default value for `newExpense` of type `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 .uiInputDateTime+.datePicker-openIcon {
    position: absolute;
    left: 90%;
    top: 30px;
}

.THIS .uiInputDefaultError li {
```

```
    list-style: none;
}
```

> 📝 **Note:** `THIS` is a keyword that adds namespacing to CSS to prevent any conflicts with another component's styling. 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.

   This step adds `<c:form />` to the markup. If you're using a namespace, you can use `<myNamespace:form />` instead.

```html
<aura:application>
    <ltng:require styles="/resource/bootstrap"/>

    <div class="bootstrap-sf1">
        <div class="navbar navbar-inverse">
         <div class="navbar-header">
        <a href="#" class="navbar-brand">My Expenses</a>
         </div>
         </div>
         <div class="container">
             <c:form />
         </div>
    </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` corresponds to a text field. The `aura` namespace includes many components for core framework functionality, like `aura:iteration` as used in this step.

SEE ALSO:

Component Markup

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

        // Perform isAccessible() check here
        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 154.

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.

```
<aura:component controller="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 99 and Sharing JavaScript Code in a Component Bundle on page 135.

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.

```
({
  getExpenses: function(component) {
        var action = component.get("c.getExpenses");
        var self = this;
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (component.isValid() && state === "SUCCESS") {
                component.set("v.expenses", response.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];

          //If you're using a namespace, use e.myNamespace__Amount__c instead
          total += e.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 `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 155.

6. Save your changes and reload your browser.

You should see the expense records created in Create an Expense Object on page 11. 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 15.

🔲 **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.

SEE ALSO:

[CRUD and Field-Level Security (FLS)](#)

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.

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.

   📝 Note: Use the API name of the fields to bind the field values. For example, if you're using a namespace, you must use `{!v.expense.myNamespace__Amount__c}` instead of `{!v.expense.Amount__c}`.

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

  <!-- Color the item blue if the expense is reimbursed -->

  <!-- If you registered a namespace, use v.expense.myNamespace__Reimbursed__c instead.
 -->
   <div class="{!v.expense.Reimbursed__c == true
                   ? 'alert alert-success' : 'alert alert-warning'}">
      <a aura:id="expense" href="{!'/' + v.expense.Id}">
          <h3>{!v.expense.Name}</h3>
      </a>

    <!-- If you registered a namespace,
        replace the following values with v.expense.myNamespace__fieldName__c instead
-->
      <p>Amount:
          <ui:outputNumber value="{!v.expense.Amount__c}" format=".00"/>
      </p>
      <p>Client:
          <ui:outputText value="{!v.expense.Client__c}"/>
      </p>
      <p>Date:
          <ui:outputDateTime value="{!v.expense.Date__c}" />
      </p>
      <p>Reimbursed?
          <ui:inputCheckbox value="{!v.expense.Reimbursed__c}" click="{!c.update}"/>
      </p>
  </div>
</aura:component>
```

Instead of using `{!expense.Amount__c}`, you're now using `{!v.expense.Amount__c}`. This expression accesses the `expense` object and the `amount` values on it.

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

4. 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.Client__c}, {!expense.Amount__c}, {!expense.Date__c},
 {!expense.Reimbursed__c}</p>
</aura:iteration>
```

Replace it with an `aura:iteration` tag that uses the `expenseList` component.

```
<aura:iteration items="{!v.expenses}" var="expense">
<!--If you're using a namespace, use myNamespace:expenseList instead-->
    <c: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.

5. 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 49.

SEE ALSO:

Component Attributes

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.

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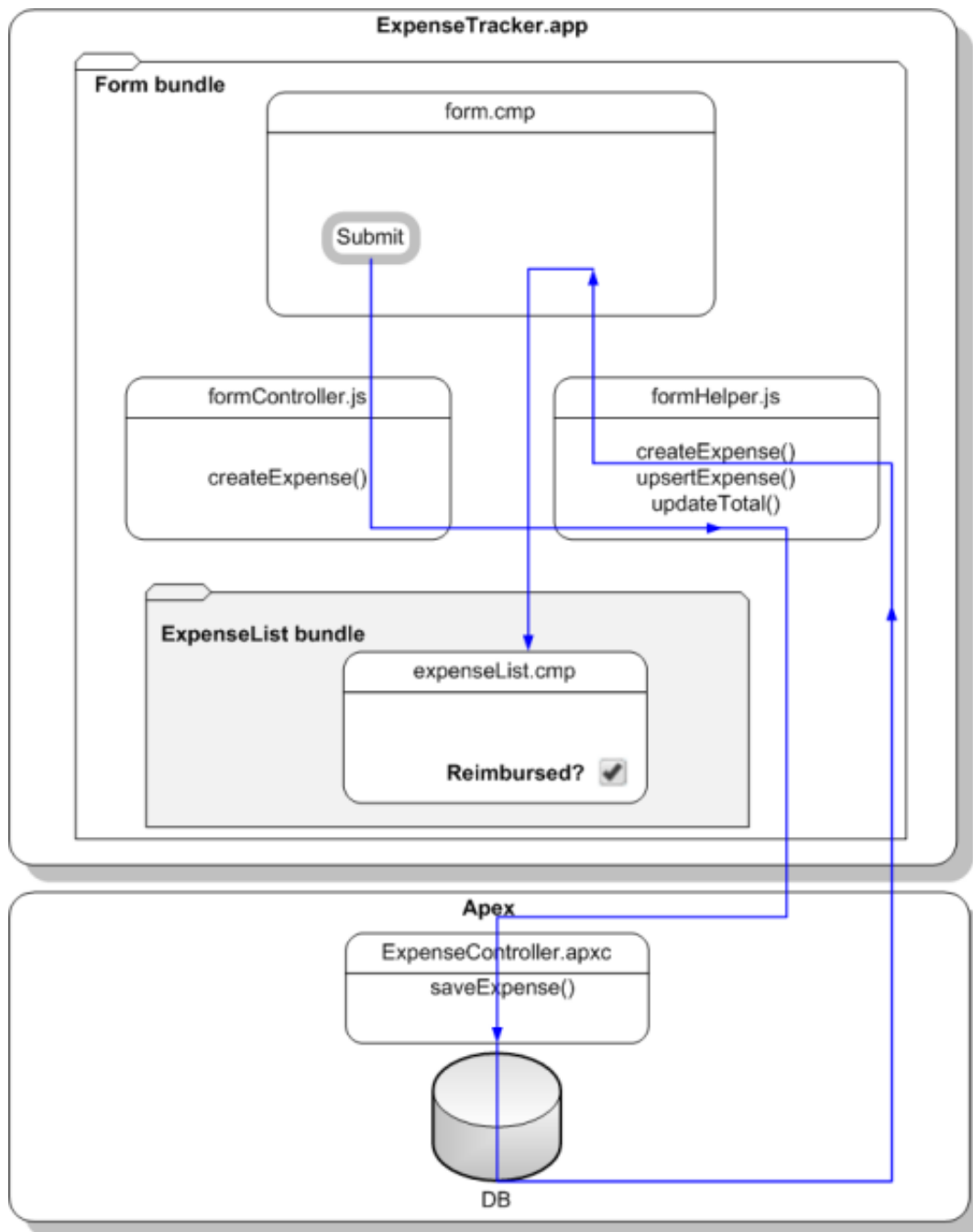
First, update the Apex controller with a new method that inserts or updates the records.

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

```
@AuraEnabled
public static Expense__c saveExpense(Expense__c expense) {

    // Perform isUpdateable() check here
    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 amount field 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 140.

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 15. `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. In `formHelper.js`, 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 182.

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
CRUD and Field-Level Security (FLS)

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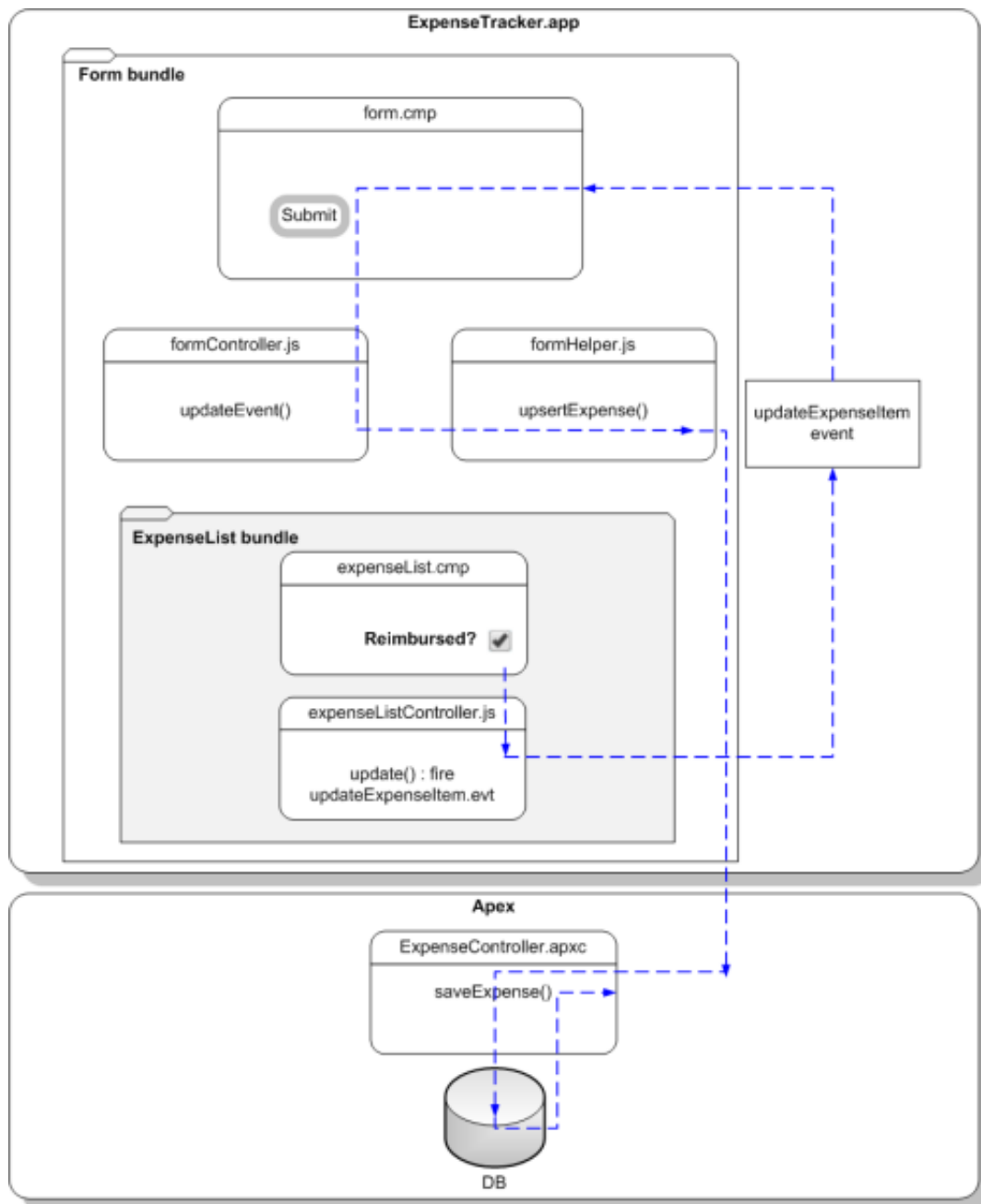
Let's start by creating the event and its handler before firing it and handling the event in the parent component.

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.

    ```
    <aura:event type="APPLICATION">
        <!-- If you're using a namespace, use myNamespace.Expense__c instead. -->
    ```

```
    <aura:attribute name="expense" type="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.

```
({
    update: function(component, evt, helper) {
      var expense = component.get("v.expense");
      //If you're using a namespace, use e.myNamespace:updateExpenseItem instead
      var updateEvent = $A.get("e.c: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.

```
<aura:handler event="c: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" , Client__c : "ABC Co." , Reimbursed__c : true , CreatedDate : "2014-08-12T20:53:09.000Z" , 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 118.

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 58. To package and distribute your app on AppExchange, see Distributing Applications and Components on page 180.

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.

28
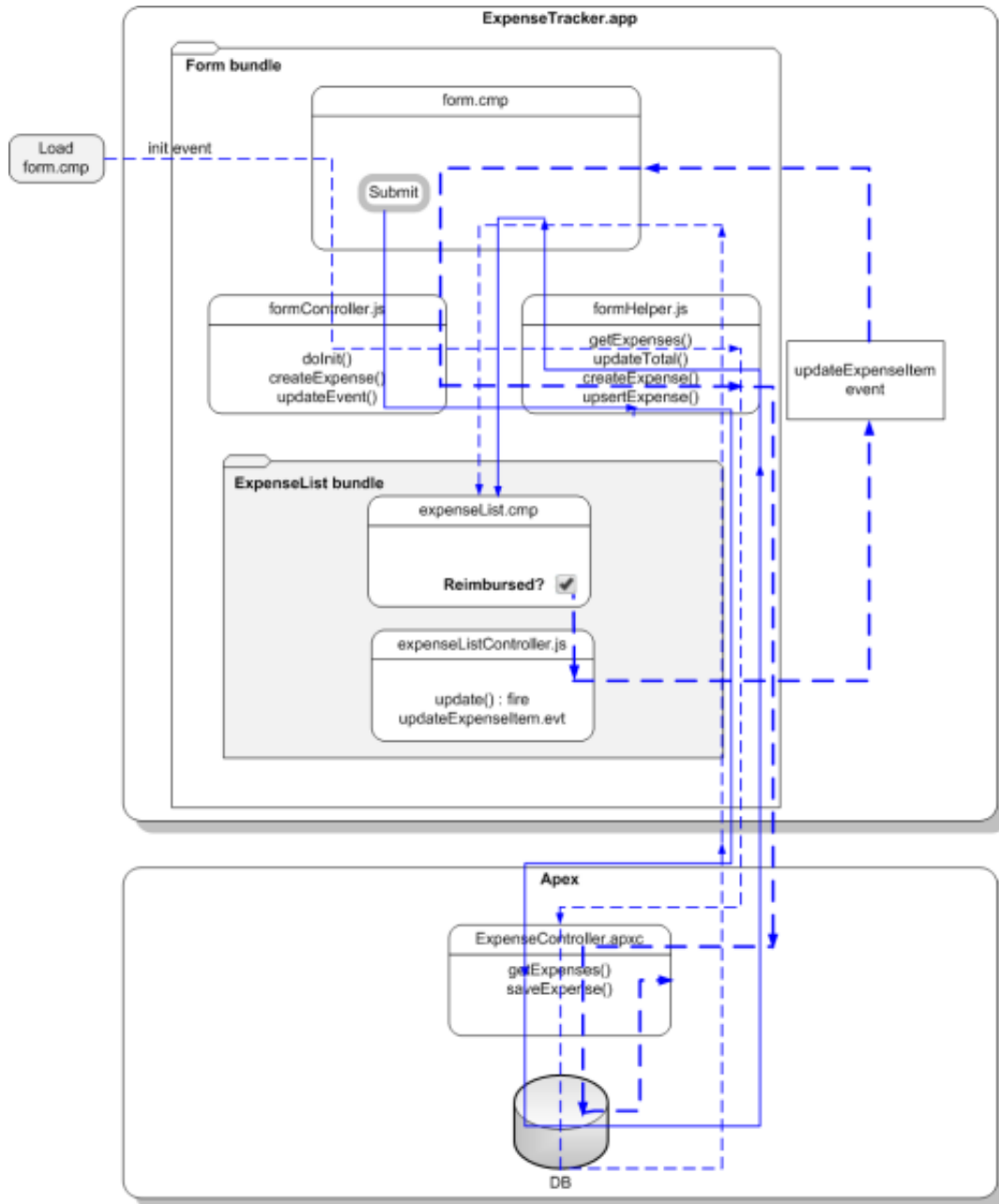
When `form.cmp` is initialized, the `init` handler triggers the `doInit` client-side controller, which calls the `getExpenses` helper function. `getExpenses` calls the `getExpenses` server-side controller to load the expenses. The callback sets the expenses data on the `v.expenses` attribute and calls `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.

Create a Component for Salesforce1

Create a component that loads contacts data and interacts with Salesforce1.
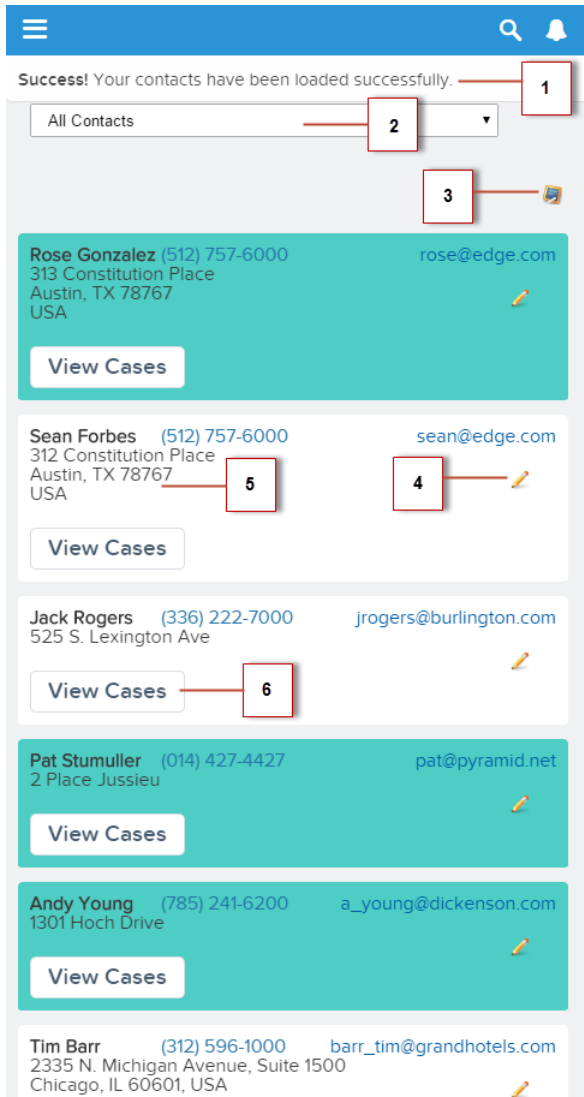
The component has these features.

- Displays a toast message (1) when all contacts are loaded successfully
- Use a nested component that displays all contacts or displays all primary contacts that are colored green when the input select value (2) is changed
- Opens the create record page to create a new contact when the Create New (3) icon is clicked
- Opens the edit record page to update the selected contact when the Edit Contact (4) icon is clicked
- Navigates to the record when the contact name (5) is clicked
- Navigates to a map when the mailing address (5) is clicked

- Navigates to related cases when the View Cases button (6) is clicked

You'll create the following resources.

Resource	Description
Contacts Bundle	
contacts.cmp	The component that loads contact data
contactsController.js	The client-side controller actions that loads contact data, handles input select change event, and opens the create record page
contactsHelper.js	The helper function that retrieves contact data and display toast messages based on the loading status
contacts.css	The styles for the component
contactList Bundle	
contactList.cmp	The contact list component
contactListController.js	The client-side controller actions that opens the edit record page, and navigates to a contact record, related cases, and map of contact address
contactList.css	The styles for the component
Apex Controller	
ContactController.apxc	The Apex controller that queries the contact records

Optional: Install the Contact List App

If you want to skip over the quick start tutorial, you can install the Contact List app as a package.

This packaged app is useful if you want to learn about the Lightning app without going through the quick start tutorial. If you're new to Lightning components, we recommend that you go through the quick start tutorial.

📝 **Note:** Install the package in an org that doesn't have any of the objects with the same API name as the quick start objects.

To install the Contact List app:

1. Click the installation URL link: https://login.salesforce.com/packaging/installPackage.apexp?p0=04to00000003pLQ
2. Log in to your organization by entering your username and password.
3. On the Package Installation Details page, click **Continue**.
4. Click **Next**, and on the Security Level page click **Next**.
5. Click **Install**.
6. Click **Deploy Now** and then **Deploy**.

When the installation completes, you can add the custom Level field to the contact page layout and select the **Contacts** tab on the user interface to add new contact records.

You'll see the Contact List menu item on the Salesforce1 navigation menu. If you don't see the menu item in Salesforce1, add it by going to **Mobile Administration** > **Mobile Navigation**.
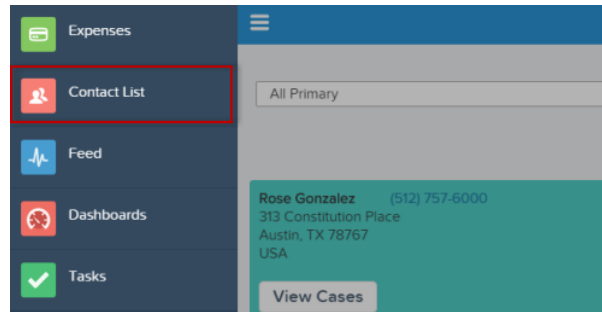
Next, you can modify the code in the Developer Console or explore the app in Salesforce1.

> 📝 **Note:** To delete the package, click **Setup** > **Installed Package**, then uninstall and delete the package.

Load the Contacts

Create an Apex controller and load your contacts.

Your organization must have existing contact records for this tutorial. This tutorial uses a custom picklist field, `Level`, which is represented by the API name `Level__c`. This field contains three picklist values: `Primary`, `Secondary`, and `Tertiary`.

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

    If you're using a namespace in your organization, replace `Level__c` with `myNamespace__Level__c`.

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

            //Add isAccessible() check
            return contacts;
        }

        @AuraEnabled
        // Retrieve all primary contacts
        public static List<Contact> getPrimary() {
            List<Contact> primaryContacts =
                    [SELECT Id, Name, MailingStreet, Phone, Email, Level__c FROM Contact WHERE
     Level__c = 'Primary'];

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

    `getPrimary()` returns all contacts whose `Level__c` field is set to `Primary`.

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

```
<aura:component>
    <aura:attribute name="contact" type="Contact"/>
    <!-- If you're using a namespace, use {!v.contact.myNamespace__Level__c} instead
-->
    <div class="{!v.contact.Level__c == 'Primary'
                ? 'row primary' : 'row '}" >

        <!-- Display a contact name
             and navigate to record when the name is clicked -->
        <div class="col-sm-4">
            <ui:outputText value="{!v.contact.Name}" click="{!c.gotoRecord}"/>
        </div>
        <div class="col-sm-4">
            <ui:outputEmail value="{!v.contact.Email}"/>
            <ui:outputPhone value="{!v.contact.Phone}"/>

            <!-- Display the edit record page when the icon is clicked -->
            <div onclick="{!c.editRecord}">
                <img src="/img/icon/custom51_100/pencil16.png" alt="Edit Contact"
title="Edit Contact" />
            </div>
        </div>
        <div class="col-sm-4">
            <ui:outputTextArea aura:id="address" value="{!v.contact.MailingStreet}"
click="{!c.navigate}"/>
        </div>

         <!-- Navigate to the related list when the button is clicked -->
         <ui:button label="View Cases" press="{!c.relatedList}"/>
    </div>
</aura:component>
```

3. In the **contactList** sidebar, click **STYLE** to create a new resource named contactList.css. Replace the placeholder code with the following code and then save.

```
.THIS.primary{
    background: #4ECDC4  !important;
}

.THIS .uiOutputText {
   width: 25%;
   float: left;
   font-weight: bold;
}

.THIS .uiOutputPhone {
    color: #2574A9;
}

.THIS .uiOutputTextArea {
    float:clear;
}
```

```css
.THIS .uiOutputEmail {
    float: right;
}


.THIS .uiButton {
    margin-top: 20px !important;
}
```

4. Click **File** > **New** > **Lightning Component**, and then enter *contacts* for the Name field in the New Lightning Bundle popup window. This creates a new component, contacts.cmp. Enter this code and then save. If you're using a namespace in your organization, replace ContactController with myNamespace.ContactController.

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

    <!-- Handle loading events by displaying a spinner -->
    <aura:handler event="aura:waiting" action="{!c.showSpinner}"/>
    <aura:handler event="aura:doneWaiting" action="{!c.hideSpinner}"/>

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

    <!-- Create a drop-down list with two options -->
    <ui:inputSelect aura:id="selection" change="{!c.select}">
        <ui:inputSelectOption text="All Contacts" label="All Contacts"/>
        <ui:inputSelectOption text="All Primary" label="All Primary"/>
    </ui:inputSelect>

    <div class="icons">
      <img src="/img/icon/custom51_100/chalkboard16.png" alt="Create New" title="Create
 New" onclick="{!c.createRecord}"/>
    </div>
    <div><center><ui:spinner aura:id="spinner"/></center></div>

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

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

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

    showSpinner : function (component, event, helper) {
        var spinner = component.find('spinner');
```

```
            var evt = spinner.get("e.toggle");
            evt.setParams({ isVisible : true });
            evt.fire();
        },

        hideSpinner : function (component, event, helper) {
            var spinner = component.find('spinner');
            var evt = spinner.get("e.toggle");
            evt.setParams({ isVisible : false });
            evt.fire();
        },//Delimiter for future code
})
```

6. In the **contacts** sidebar, click **HELPER** to create a new resource named `contactsHelper.js`. Replace the placeholder code with the following code and then save.

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

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

7. In the **contacts** sidebar, click **STYLE** to create a new resource named `contacts.css`. Replace the placeholder code with the following code and then save.

```
.THIS.row {
    background: #fff;
    max-width:90%;
    border-bottom: 2px solid #f0f1f2;
```

```
    padding: 10px;
    margin-left: 2%;
    margin-bottom: 10px;
    min-height: 70px;
    border-radius: 4px;
}

.THIS.uiInputSelect {
    width: 80%;
    padding-left: 10px;
    min-height: 28px;
    margin-bottom: 20px;
    margin-left: 20px;
    margin-top: 40px;
}

.THIS img {
    float: right;
    padding-right:5%;
    padding-top: 20px;
}

.THIS.icons {
    height:60px;
}
```

8. Create a new Lightning Component tab by following the steps on Adding Lightning Components to Salesforce1 on page 58. Make sure you include the component in the Salesforce1 navigation menu.

Finally, you can go to the Salesforce1 mobile browser app to check your output. When your component is loaded, you should see a toast message that indicates your contacts are loaded successfully.

Next, we'll wire up the other events so that your input select displays either all contacts or only primary contacts that are colored green. We'll also wire up events for opening the create record and edit record pages, and events for navigating to a record and a URL.

Fire the Events

Fire the events in your client-side controller or helper functions. The `force` events are handled by Salesforce1.

This demo builds on the contacts component you created in Load the Contacts on page 32.

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

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

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

    navigate : function(component, event, helper) {
        // Navigate to an external URL
        var address = component.find("address").get("v.value");
        var urlEvent = $A.get("e.force:navigateToURL");
        urlEvent.setParams({
          "url": 'https://www.google.com/maps/place/' + address
        });
        urlEvent.fire();
    },

    relatedList : function (component, event, helper) {
        // Navigate to the related cases
        var relatedListEvent = $A.get("e.force:navigateToRelatedList");
        relatedListEvent.setParams({
            "relatedListId": "Cases",
            "parentRecordId": component.get("v.contact.Id")
        });
        relatedListEvent.fire();
    }
})
```

2. Refresh the Salesforce1 mobile browser app, and click these elements to test the events.

- Contact name: `force:navigateToSObject` is fired, which updates the view with the contact record page. The contact name corresponds to the following component.

```
<ui:outputText value="{!v.contact.Name}" click="{!c.gotoRecord}"/>
```

- Edit Contact icon: `force:editRecord` is fired, which opens the edit record page. The Edit Contact icon corresponds to the following component.

```
<div onclick="{!c.editRecord}">
    <img src="/img/icon/custom51_100/pencil16.png" alt="Edit Contact" title="Edit
Contact" />
</div>
```

- Address: `force:navigateToURL` is fired, which opens Google Maps with the provided `url`. The address corresponds to the following component.

```
<div class="col-sm-4">
    <ui:outputTextArea aura:id="address" value="{!v.contact.MailingStreet}"
click="{!c.navigate}"/>
</div>
```

3. Open `contactsController.js`. After the `hideSpinner` controller, enter this code and then save.

```
createRecord : function (component, event, helper) {
    // Open the create record page
    var createRecordEvent = $A.get("e.force:createRecord");
    createRecordEvent.setParams({
        "entityApiName": "Contact"
    });
    createRecordEvent.fire();
},

select : function(component, event, helper){
    // Get the selected value of the ui:inputSelect component
    var selectCmp = component.find("selection");
    var selectVal = selectCmp.get("v.value");

    // Display all primary contacts or all contacts
    if (selectVal==="All Primary"){
        var action = component.get("c.getPrimary");
        action.setCallback(this, function(response){
            var state = response.getState();
            if (component.isValid() && state === "SUCCESS") {
                component.set("v.contacts", response.getReturnValue());
            }
        });
        $A.enqueueAction(action);
    }
    else {
        // Return all contacts
        helper.getContacts(component);
    }
}
```

Notice that if you pull down the page and release it, the page refreshes all data in the view. This is similar to running `$A.get('e.force:refreshView').fire();`.

Now you can test your components by clicking on the areas highlighted in Create a Component for Salesforce1 on page 30.

For an example on creating a standalone app that can be used independent of Salesforce1, see Create a Standalone Lightning App on page 8.

CHAPTER 3   Components

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.

Use the Developer Console to create components.

SEE ALSO:

   Using the Developer Console

39

Component Markup

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

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

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

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

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

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

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

Use the Developer Console to create components.

`aura:component` 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.

SEE ALSO:

Using the Developer Console

Component Access Control

Client-Side Rendering to the DOM

Dynamically Creating Components

Component Namespace

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

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

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

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

Using the Default Namespace in Organizations with No Namespace Set

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

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

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

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

- References to custom objects
- References to custom fields on standard and custom objects
- References to Apex controllers
- Components that you create dynamically in Apex

> Note: Optionally, components that you create dynamically can use the `c` namespace if you prefer to specify a namespace.

See for examples of all of the preceding items.

Using Your Organization's Namespace

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

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

- References to components that you've created
- References to events that you've defined
- References to custom objects
- References to custom fields on standard and custom objects
- References to Apex controllers
- Components that you create dynamically in Apex

> Note: Support for the `c` namespace in organizations that have set a namespace prefix is incomplete. The following items can use the `c` namespace if you prefer to use the shortcut, but it's not currently a recommended practice.
>
> - References to components that you've created when used in Lightning markup, but not in expressions or JavaScript
> - References to events that you've defined when used in Lightning markup, but not in expressions or JavaScript
> - References to custom objects when used in component and event `type` and `default` system attributes, but not in expressions or JavaScript

Using a Namespace in or from a Managed Package

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

Creating a Namespace in Your Organization

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

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

To register a namespace prefix:

1. From Setup, click **Create** > **Packages**.

2. Click **Edit**.

   > Note: This button doesn't appear if you've already configured your developer settings.

3. Review the selections that are required for configuring developer settings, and then click **Continue**.

4. Enter the namespace prefix you want to register.

5. Click **Check Availability** to determine if the namespace prefix is already in use.

6. If the namespace prefix that you entered isn't available, repeat the previous two steps.

7. Click **Review My Selections**.

8. Click **Save**.

Namespace Usage Examples and Reference

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

Examples are provided for the following.

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

Organizations with No Namespace Prefix Set

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

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

Referenced Item	Example

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

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

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

Custom field in an expression	`<ui:inputNumber value="{!v.newExpense.Amount__c}"    label=…` `/>`

Custom field in a JavaScript function	`updateTotal: function(component) {`
	`    …`
	`    for(var i = 0 ; i < expenses.length ; i++){`
	`        var exp = expenses[i];`
	`        total += exp.Amount__c;`
	`    }`
	`    …`
	`}`

Component created dynamically in a JavaScript function	`var myCmp = $A.services.component.newComponent({`
	`    componentDef : {`
	`        descriptor : "markup://c:myComponent"`
	`    }`
	`});`

Interface comparison in a JavaScript function	`aCmp.isInstanceOf("c:myInterface")`
Event registration	`<aura:registerEvent type="c:updateExpenseItem"    name=… />`
Event handler	`<aura:handler event="c:updateExpenseItem" action=…   />`
Explicit dependency	`<aura:dependency resource="markup://c:myComponent" />`
Application event in a JavaScript function	`var updateEvent = $A.get("e.c:updateExpenseItem");`

Component created dynamically in Apex	`Cmp.expenseList expList = new Cmp.expenseList();`
	
	`Cmp.c.expenseList expList = new Cmp.c.expenseList();`
	
	The previous two lines of code are equivalent. The default namespace is supported but optional in this context.

Organizations with a Namespace Prefix

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

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

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

Apex controller	`<aura:component  controller="yournamespace.ExpenseController">`
Custom object in attribute data type	`<aura:attribute name="expenses"` `type="yournamespace.Expense__c[]" />`

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

Custom field in an expression	`<ui:inputNumber` `value="{!v.newExpense.yournamespace__Amount__c}" label=… />`

Custom field in a JavaScript function	`updateTotal: function(component) {`
	`    …`
	`    for(var i = 0 ; i < expenses.length ; i++){`
	`        var exp = expenses[i];`
	`        total += exp.yournamespace__Amount__c;`
	`    }`
	`    …`
	`}`

Component created dynamically in a JavaScript function	`var myCmp = $A.services.component.newComponent({`
	`    componentDef : {`
	`        descriptor : "markup://yournamespace:myComponent"`
	`    }`
	`});`

Interface comparison in a JavaScript function	`aCmp.isInstanceOf("yournamespace:myInterface")`
Event registration	`<aura:registerEvent  type="yournamespace:updateExpenseItem"` `name=… />`
Event handler	`<aura:handler  event="yournamespace:updateExpenseItem"` `action=… />`

Referenced Item	Example
Explicit dependency	`<aura:dependency resource="markup://`**`yournamespace:myComponent`**`" />`
Application event in a JavaScript function	`var updateEvent = $A.get(`**`"e.yournamespace:updateExpenseItem");`**
Component created dynamically in Apex	`Cmp`**`.yournamespace.expenseList`**` expList = new Cmp`**`.yournamespace.expenseList`**`();`

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 39 aura:application on page 185
CSS Styles	`sample.css`	Styles for the component.	CSS in Components on page 47
Controller	`sampleController.js`	Client-side controller methods to handle events in the component.	Handling Events with Client-Side Controllers on page 99
Design	`sample.design`	Required for components used in the Lightning App Builder or Lightning Pages.	Configure Components for Lightning Pages and the Lightning App Builder
Documentation	`sample.auradoc`	A description, sample code, and one or multiple references to example components	Providing Component Documentation on page 61
Renderer	`sampleRenderer.js`	Client-side renderer to override default rendering for a component.	Client-Side Rendering to the DOM on page 136
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 135
SVG File	sample.svg	Custom icon resource for components used in the Lightning App Builder.	Configure Components for Lightning Pages and the Lightning App Builder on page 59

45

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")` in your client-side controller, 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`.

Global IDs

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

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

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

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

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

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

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

> 📝 **Note:** 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`.

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 286.

Unescaping HTML

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

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

`{!expression}` is the framework's expression syntax. For more information, see Expressions on page 64.

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 128.

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

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

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

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

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

Attribute 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}"/>`.

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

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

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

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

Attribute Validation

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

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

SEE ALSO:

Supported aura:attribute Types

Expressions

Component Composition

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

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

Here is the source for `helloHTML.cmp`.

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

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

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

CSS source

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

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

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

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

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

Output

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

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

Here is the source for `helloAttributes.cmp`.

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

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

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

    <docsample:helloHTML/>

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

Output

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

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

Hello component composition!

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

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

Attribute Passing

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

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


    <docsample:helloHTML/>


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

Output

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

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

Hello passed attribute!

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

Definitions versus Instances

In object-oriented programming, there's a difference between a class and an instance of that class. Components have a similar concept. When you create a `.cmp` resource, you are providing the definition (class) of that component. When you put a component tag in a `.cmp`, 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 `docsample:helloAttributes` with a different attribute value. The two instances of the `docsample:helloAttributes` component have different values for their `whom` attribute.

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

51

```
    <docsample:helloHTML/>

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

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

Output

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

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

Hello passed attribute! Hello separate instance!

Component Body

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

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

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

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

Setting the Body Content

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

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

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

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

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

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

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

This is a shortcut for:

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

Accessing the Component Body

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

SEE ALSO:

aura:set

Working with a Component Body in JavaScript

Component Facets

A facet is any attribute of type `Aura.Component[]`. The `body` attribute is an example of a facet.

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

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>
```

Note that `aura:set` sets the value of an attribute inherited from the super component, but you don't need to use `aura:set` if you're setting the value of `v.body`.

SEE ALSO:

Component Body

Best Practices for Conditional Markup

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

`<aura:if>` versus `<aura:renderIf>`

`<aura:if>` is more lightweight than `<aura:renderIf>` as it only creates and renders the markup in its body or in the `else` attribute. Always try `<aura:if>` first when you want conditional markup.

Only consider using `<aura:renderIf>` if you expect to show the markup for both the true and false states, and it would require a server round trip to create the components that aren't initially rendered.

Here's a quick comparison of `<aura:if>` versus `<aura:renderIf>`.

	`<aura:if>`	`<aura:renderIf>`
Displaying	Creates and displays only one branch	Creates both branches but only displays one
Switching condition	Unrenders and destroys the current branch. Creates and displays the other branch.	Unrenders the current branch and renders the other branch
Empty branch	Creates a DOM placeholder	Creates a DOM placeholder

Consider Alternatives to Conditional Markup

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

You want to toggle visibility

Don't use `<aura:if>` or `<aura:renderIf>` tags to toggle markup visibility. Use CSS instead. See Dynamically Showing or Hiding Markup on page 150.

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

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

Consider alternatives, such as using JavaScript logic in an `init` event handler instead. See Invoking Actions on Component Initialization on page 146.

SEE ALSO:

Conditional Expressions

Using Labels

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

IN THIS SECTION:

Using Custom Labels

Use custom labels in Lightning components with the `$Label` global value provider.

Input Component Labels

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

Setting Label Values via a Parent Attribute

Setting label values via a parent attribute is useful if you want control over labels in child components.

`Using Custom Labels`

Use custom labels in Lightning components with the `$Label` global value provider.

Custom labels are custom text values that can be translated into any language Salesforce supports. Custom labels enable developers to create multilingual applications by automatically presenting information (for example, help text or error messages) in a user's native language.

To create custom labels, from Setup, click **Create** > **Custom Labels**.

Use this syntax to access custom labels in Lightning components:

- `$Label.c.`***`labelName`*** for the default namespace
- `$Label.`***`namespace.labelName`*** if your org has a namespace

Here are some examples.

Label in a markup expression using the default namespace

`{!$Label.c.`***`labelName`***`}`

Label in JavaScript code if your org has a namespace

`$A.get("`$Label.***`namespace.labelName`***`")`

> 📝 Note: Custom labels referenced in a Lightning component are not automatically added when you create a package containing the Lightning component. To include a custom label in a package, another member of the package, such as a Visualforce page, must reference the custom label.

SEE ALSO:

Global Value Providers

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.

SEE ALSO:

[Supporting 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 in Salesforce1

You must opt in to enable Lightning components in Salesforce1. Lightning components in standalone apps (`.app` resources) are available by default.

1. From Setup, click **Develop** > **Lightning Components**.

2. Select the `Enable Lightning Components` checkbox.

> ⚠ Warning: This beta version of Lightning components in Salesforce1 doesn't support Force.com Canvas apps in Salesforce1. 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.

> 📝 Note: Lightning components in standalone apps (`.app` resources) are generally available. Integrating Lightning components with Salesforce1 is in a beta release that is production quality but has some limitations.

Lightning components in standalone apps are available by default. You can enable Lightning components in Salesforce1 from Setup by navigating to **Develop** > **Lightning Components**.

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

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

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

Use the Developer Console to create Lightning components.

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*.

Configure Components for Lightning Pages and the Lightning App Builder

There are two adjustments you must make before you can use your custom Lightning components in either Lightning Pages or the Lightning App Builder.

Add a New Interface to Your Component

To appear in the Lightning App Builder or a Lightning Page, a component must implement the `flexipage:availableForAllPageTypes` interface.

Here's the sample code for a simple "Hello World" component.

```
<aura:component implements="flexipage:availableForAllPageTypes">
    <aura:attribute name="greeting" type="String" default="Hello" />
    <aura:attribute name="subject" type="String" default="World" />

    <div style="box">
      <span class="greeting">{!v.greeting}</span>, {!v.subject}!
    </div>
</aura:component>
```

Add a Design Resource to Your Component Bundle

You must include a design resource in the component bundle to make your Lightning component usable in Lightning Pages and the Lightning App Builder. A design resource describes the design-time behavior of a Lightning component—information that visual tools need to allow adding the component to a page or app.

To make a Lightning component attribute available for administrators to edit in the Lightning App Builder, add a `design:attribute` node for the attribute into the design resource.

An attribute marked as required in the component definition automatically appears for users in the Lightning App Builder, unless it has a default value assigned to it. Required attributes with default values and attributes not marked as required in the component definition must be specified in the design resource or they won't appear for users.

Here's the design resource that goes in the bundle with the "Hello World" component.

```
<design:component label="Hello World">
    <design:attribute name="subject" label="Subject" description="Name of the person you
want to greet" />
    <design:attribute name="greeting" label="Greeting" />
</design:component>
```

To render a field as a picklist, add a `datasource` onto the attribute in the design resource, like this:

```
<design:attribute name="Name" datasource="value1,value2,value3" />
```

Any string attribute with a `datasource` in a design resource is treated as a picklist. How the Lightning App Builder renders it depends on the data type that you define in the component. For example:

- `<aura:attribute name="Name" type="String" />` renders as a picklist
- `<aura:attribute name="Name" type="String[]" />` renders as a multi-select picklist

A design resource supports only attributes of type `int`, `string`, or `boolean`.

Design resources must be named `componentName.design`.

Optional: Add an SVG Resource to Your Component Bundle

You can use an SVG resource to define a custom icon for your component when it appears in the Lightning App Builder's component pane. Just include it in the component bundle.

Here's a simple red circle SVG resource to go with the "Hello World" component.

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     width="400" height="400">
  <circle cx="100" cy="100" r="50" stroke="black"
    stroke-width="5" fill="red" />
</svg>
```

SVG resources must be named `componentName.svg`.

SEE ALSO:

Component Bundles

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
   Using Object-Oriented Development
   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

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

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

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

expression is a placeholder for the expression.

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

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

📝 **Note:** If you're familiar with other languages, you may be tempted to read the `!` as the "bang" operator, which negates boolean values in many programming languages. In 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.

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

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

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

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

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

Dynamic Output in Expressions

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

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

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

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

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

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

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

SEE ALSO:

Component Attributes

Value Providers

Conditional Expressions

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

Ternary Operator

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

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

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

Using `<aura:if>` for Conditional Markup

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

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

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

SEE ALSO:

Best Practices for Conditional Markup

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 the values on that attribute using the same dot notation.

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

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

SEE ALSO:

Dynamic Output in Expressions

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 46.
- `$Browser`—See $Browser on page 66.
- `$Label`—See Using Custom Labels on page 55.
- `$Locale`—See $Locale on page 67.

$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 returns true or false depending on the operating system and device of the browser where you are rendering the component.

Component source

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

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

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

$Locale

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

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

Attribute	Description	Sample Value
country	The ISO 3166 representation of the country code.	"US", "DE", "GB"
currency	The currency symbol.	"$"
currencyCode	The ISO 4217 representation of the currency code.	"USD"
decimal	The decimal separator.	"."
grouping	The grouping separator.	","
language	The language code.	"en", "de", "zh"
langLocale	The locale ID.	"en_US", "en_GB"
timezone	The time zone ID.	"America/Los_Angeles", "America/New_York"
variant	The vendor and browser-specific code.	"WIN", "MAC", "POSIX"

Number and Date Formatting

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

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

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

Component source

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

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

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

SEE ALSO:

[Localization](#)

Expression Evaluation

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

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

Action Methods

Expressions are also used to provide action methods for user interface events: `onclick`, `onhover`, and any other component attributes beginning with "`on`". Some components simplify assigning actions to user interface events using other attributes, such as the `press` attribute on `<ui:button>`.

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

Assigning action methods via expressions allows you to assign them conditionally, based on the state of the application or user interface. For more information, see [Conditional Expressions](#) on page 65.

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

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

Expression Operators Reference

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

Arithmetic Operators

Expressions based on arithmetic operators result in numerical values.

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

Numeric Literals

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

String Operators

Expressions based on string operators result in string values.

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

String Literals

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

Literal	Usage	Description
string	`'hello world'`	Literal strings must be enclosed in single quotation marks. Double quotation marks are reserved for enclosing attribute values, and must be escaped in strings.

Literal	Usage	Description
\<escape>	`'\n'`	Whitespace characters: • `\t` (tab) • `\n` (newline) • `\r` (carriage return) Escaped characters: • `\"` (literal ") • `\'` (literal ') • `\\` (literal \)
Unicode	`'\u####'`	A Unicode code point. The # symbols are hexadecimal digits. A Unicode literal requires four digits.
null	`null`	A literal null string. Matches the explicit null value and strings with an undefined value.

Comparison Operators

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

Operator	Alternative	Usage	Description

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

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

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

Operator	Alternative	Usage	Description
>	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 this negation operator to return the logical negation of a value, for example, `{!!true}` returns `false`.

Logical Literals

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

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

Conditional Operator

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

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

SEE ALSO:

Expression Functions Reference

Expression Functions Reference

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

Math Functions

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

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

Function	Alternative	Usage	Description	Corresponding Operator
`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.	`+`

Informational Functions

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

Comparison Functions

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

Function	Usage	Description	Corresponding Operator
`equals`	`equals(1,1)`	Returns `true` if the specified arguments are equal. The arguments can be any data type.	`==` or `eq`

Function	Usage	Description	Corresponding Operator
`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   UI Components

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

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`.

Complex, Interactive Components

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

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

Input Control Components

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

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

Visual Components

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

Type	Key Components	Description
Field-level error	`ui:inputDefaultError`	An error message that is displayed when an error occurs
Spinner	`ui:spinner`	A loading spinner

Field Components

The following components enables you to enter or display values.

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

Type	Key Components	Description
	`ui:outputNumber`	Displays a number
Range	`ui:inputRange`	An input field for entering a value within a range
Rich Text	`ui:inputRichText`	An input field for entering rich text
	`ui:outputRichText`	Displays rich text
Text	`ui:inputText`	An input field for entering a single line of text
	`ui:outputText`	Displays text
Text Area	`ui:inputTextArea`	An input field for entering multiple lines of text
	`ui:outputTextArea`	Displays a read-only text area
URL	`ui:inputURL`	An input field for entering a URL
	`ui:outputURL`	Displays a clickable URL

SEE ALSO:

Using the UI Components

Components

Component Bundles

UI Events

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

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

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

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

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

For all available events on all components, see the Component Reference on page 189.

Value Binding for Browser Events

Any changes to the UI are reflected in the component attribute, and any change in that attribute is propagated to the UI. When you load the component, the value of the input elements are initialized to those of the component attributes. Any changes to the user input

causes the value of the component variable to be updated. For example, a `ui:inputText` component can contain a value that's bound to a component attribute, and the `ui:outputText` component is bound to the same component attribute. The `ui:inputText` component listens to the `onkeyup` browser event and updates the corresponding component attribute values.

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

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

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

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

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

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

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

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

Using the UI Components

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

📝 Note: For all available component attributes and events, see the component reference at `https://<mySalesforceInstance>.lightning.force.com/auradocs/reference.app`, where `<mySalesforceInstance>` is the name of the instance hosting your org; for example, `na1` .

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. The `aura:id` attribute defines a unique ID that enables you to reference the component from your JavaScript code using `cmp.find("myID");`.

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

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

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

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

```
    var outName = cmp.find("nameOutput");
    outName.set("v.value", fullName);
  }
```

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

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

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

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

> 💡 **Tip:** To create and edit records in Salesforce1, use the `force:createRecord` and `force:recordEdit` events to utilize the built-in record create and edit pages.

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>
```

Binding Field Values

You can bind field values to a field in an object using expressions such as `{!v.myAttribute.Name}` or `{!v.myAttribute.namespace__MyField__c}`, and saving an input value via an Apex controller. For an example, see Create a Standalone Lightning App on page 8.

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.

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

Number fields are represented by the following components.

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

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>
```

Binding Field Values

You can bind field values to a field in an object using expressions such as `{!v.myAttribute.Name}` or `{!v.myAttribute.namespace__MyField__c}`, and saving an input value via an Apex controller. For an example, see Create a Standalone Lightning App on page 8.

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

You can style the appearance of your number field and output. In the CSS file of your component, add the corresponding class selectors. The following class selectors provide styles to the string rendering of the numbers. For example, to style the `ui:inputCurrency` component, use `.THIS .uiInputCurrency`, or `.THIS.uiInputCurrency` if it's a top-level element.

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

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

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

SEE ALSO:

Input Component Labels

Handling Events with Client-Side Controllers

Localization

CSS in Components

Text Fields

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

Text fields are represented by the following components.

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

Using the Text Fields

Text fields are typically used in a form. For example, this is a basic set up of an email field.

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

This example results in the following HTML.

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

> ✏️ **Note:** You can also use the force:navigateToURL event to make an element behave like a URL link. For more information, see force:navigateToURL on page 273.

Binding Field Values

You can bind field values to a field in an object using expressions such as `{!v.myAttribute.Name}` or `{!v.myAttribute.namespace__MyField__c}`, and saving an input value via an Apex controller. For an example, see Create a Standalone Lightning App on page 8.

Styling Your Text Fields

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

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

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

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

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

SEE ALSO:

Rich Text Fields

Input Component Labels

Handling Events with Client-Side Controllers

Localization

CSS in Components

Rich Text Fields

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

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

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

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

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

SEE ALSO:

Text Fields

Checkboxes

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

Here are several basic ways to set up a checkbox.

Checked

To select the checkbox, set `value="true"`. This example sets the inital value of the checkbox.

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

85

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

Radio Buttons

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

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

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

Selected

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

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

Disabled State

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

The previous example results in the following HTML.

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

Providing Labels using An Attribute

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

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

Working with Events

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

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

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

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

SEE ALSO:

Handling Events with Client-Side Controllers

CSS in Components

Buttons

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

- Text-only Button

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

- Image-only Button

```
<!-- 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

Drop-down Lists

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

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

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

Single Selection

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

Multiple Selection

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

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

Generating Options with `aura:iteration`

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

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

Generating Options Dynamically

Generate the options dynamically on component initialization.

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

The following client-side controller generates options using the `options` attribute on the `ui:inputSelect` component. `v.options` takes in the list of objects and converts them into list options. Although the sample code generates the options during initialization, the list of options can be modified anytime when you manipulate the list in `v.options`. The component automatically updates itself and rerenders with the new options.

```
({
    doInit : function(cmp) {
        var opts = [
            { class: "optionClass", label: "Option1", value: "opt1", selected: "true" },
            { class: "optionClass", label: "Option2", value: "opt2" },
            { class: "optionClass", label: "Option3", value: "opt3" }

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

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

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

The `InputOption` object has these parameters.

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

Working with Events

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

Styling Your Field-level Errors

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

```
.THIS.uiInputSelect {
    width: 200px;
    height: 100px;
}
```

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

SEE ALSO:
> [Handling Events with Client-Side Controllers](#)
>
> [CSS in Components](#)

Field-level Errors

Field-level errors are displayed when 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.

Alternatively, you can use `ui:message` for field-level errors by toggling visibility of the message when an error condition is met. See [Dynamically Showing or Hiding Markup](#) for more information.

Invalid password
Your password should be at least 6 alphanumeric characters long.

Working with Events

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

SEE ALSO:
> [Handling Events with Client-Side Controllers](#)
>
> [CSS in Components](#)

Menus

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

This example creates a menu with several items.

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

You can display a list of items from an object. This example displays a list of contact names in a menu using `aura:iteration`.

```
<aura:component>
    <aura:attribute name="contacts" type="String[]" default="All,Primary,Secondary"/>
    <ui:menu>
        <ui:menuTriggerLink label="Select Contact"/>
        <ui:menuList>
            <aura:iteration var="contact" items="{!v.contacts}">
                <ui:actionMenuItem label="{!contact}"/>
            </aura:iteration>
        </ui:menuList>
    </ui:menu>
</aura:component>
```

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

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

CHAPTER 6   Supporting Accessibility

When customizing these components or their sub-components, be careful in preserving code that ensures accessibility, such as the `aria` attributes. See UI Components for components you can use in your apps.

94

Accessibility Considerations

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

SEE ALSO:

  Supporting Accessibility

  Components

  Handling Events with Client-Side Controllers

Button Labels

Buttons may be designed to appear with just text, an image and text, or an image without text. To create an accessible button, use `ui:button` and set a textual label using the `label` attribute. The text is available to assistive technologies, but not visible on screen.

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

When using `ui:button`, assign a non-empty string to label attribute. These examples show how a `ui:button` should render:

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

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

SEE ALSO:

  Buttons

Help and Error Messages

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

If your code failed, check to see if `ariaDescribedby` is missing. Your component should render like this example:

```
<!-- Good: aria-describedby is used to associate error message -->
<label for="fname">Contact name</label>
<input name="" type="text" id="fname" aria-describedby="msgid">
```

```
<ul class="uiInputDefaultError" id="msgid">
 <li>Please enter the contact name</li>
</ul>
```

Audio Messages

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

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

Forms, Fields, and Labels

Input components are designed to make it easy to assign labels to form fields. Labels build a programmatic relationship between a form field and its textual label. When using a placeholder in an input component, set the `label` attribute for accessibility.

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

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

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

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

Events

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

Menus

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

This example code creates a menu with several items:

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

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

Actions

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

Radio button

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

Checkbox style

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

COMMUNICATING WITH EVENTS

CHAPTER 7   Events

In this chapter ...

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

In the 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.

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

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.

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 work properly. 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 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

Creating Server-Side Logic with Controllers

Actions and Events

The framework uses events to relay data between components, which are usually triggered by a user action. Here are some considerations for working with actions and events.

Actions

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

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

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

Events

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

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

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

1. User clicks a button or interacts with a component, triggering a browser event. For example, you want to save data from the server when the button is clicked.

2. The button click invokes a client-side JavaScript controller, which provides some custom logic before invoking a helper function.

3. The JavaScript controller invokes a helper function. Note that a helper function improves code reuse but it's optional for this example.

4. The helper function calls an Apex controller method and queues the action.

5. The Apex method is invoked and data is returned.

6. A JavaScript callback function is invoked when the Apex method completes.

7. The JavaScript callback function evaluates logic and updates the component's UI.

8. User sees the updated component.

Alternatively, consider an attribute value on a component that changes without a user action directly causing it, which then automatically fires a `change` event. When the attribute value changes, the component that registers a `change` event handles this event by invoking a JavaScript controller that contains custom logic, which could then proceed from step (3) onwards to retrieve data from the server.

SEE ALSO:

Handling Events with Client-Side Controllers

Detecting Data Changes

Calling a Server-Side Action

Events Fired During the Rendering Lifecycle

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 `docsample:compEvent` component event with one `message` attribute.

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

The component that 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="docsample: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="docsample: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 `<docsample:eventsNotifier>` component contains a `<aura:registerEvent>` tag.

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

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

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

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

103

Get the Source of a Component Event

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

SEE ALSO:

Application Events

Handling Events with Client-Side Controllers

Advanced Events Example

What is Inherited?

Component Event Example

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

1. A user clicks a button in the notifier component, `ceNotifier.cmp`.

2. The client-side controller for `ceNotifier.cmp` sets a message in a component event and fires the event.

3. The handler component, `ceHandler.cmp`, contains the notifier component, and handles the fired event.

4. The client-side controller for `ceHandler.cmp` sets an attribute in `ceHandler.cmp` based on the data sent in the event.

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

Component Event

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

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

Notifier Component

The `ceNotifier.cmp` 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.

```
<!--docsample:ceNotifier-->
<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

The `ceHandler.cmp` 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.

```
<!--docsample:ceHandler-->
<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);
    }
}
```

105

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 a `docsample:appEvent` application event with one `message` attribute.

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

The component that 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="docsample: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.docsample: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="docsample: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

SEE ALSO:

Component Events

Handling Events with Client-Side Controllers

Advanced Events Example

What is Inherited?

Application Event Example

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

1. A user clicks a button in the notifier component, `aeNotifier.cmp`.

2. The client-side controller for `aeNotifier.cmp` sets a message in a component event and fires the event.

3. The handler component, `aeHandler.cmp`, handles the fired event.

4. The client-side controller for `aeHandler.cmp` sets an attribute in `aeHandler.cmp` based on the data sent in the event.

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

Application Event

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

```
<!--docsample:aeEvent-->
<aura:event type="APPLICATION">
    <aura:attribute name="message" type="String"/>
</aura:event>
```

Notifier Component

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

```
<!--docsample:aeNotifier-->
<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

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

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

```
<!--docsample:aeHandler-->
<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

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

```
<!--docsample:aeContainer-->
<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

Here is the markup for `compEvent.evt`.

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

Application Event

Here is the markup for `appEvent.evt`.

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

Notifier Component

The `eventsNotifier.cmp` notifier component contains 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`.

```
<!--docsample:eventsNotifier-->
<aura:component>
  <aura:attribute name="parentName" type="String"/>
  <aura:registerEvent name="componentEventFired" type="docsample:compEvent"/>
  <aura:registerEvent name="appEvent" type="docsample:appEvent"/>

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

CSS source

The CSS is in `eventsNotifier.css`.

```
/* eventsNotifier.css */
.docsampleEventsNotifier {
    display: block;
```

```
    margin: 10px;
    padding: 10px;
    border: 1px solid black;
}
```

Client-side controller source

The `eventsNotifierController.js` controller fires the event.

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

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

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

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

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

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

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

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

Handler Component

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

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

  <aura:attribute name="numComponentEventsHandled" type="Integer" default="0"/>
  <aura:attribute name="numApplicationEventsHandled" type="Integer" default="0"/>
  <aura:handler event="docsample: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>
    <docsample:eventsNotifier parentName="{!v.name}"
componentEventFired="{!c.handleComponentEventFired}"/>
  </div>
</aura:component>
```

CSS source

The CSS is in `eventsHandler.css`.

```
/* eventsHandler.css */
.docsampleEventsHandler {
  display: block;
  margin: 10px;
  padding: 10px;
  border: 1px solid black;
}
```

Client-side controller source

The client-side controller is in `eventsHandlerController.js`.

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

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

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

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

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

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

115

Container Component

Here is the markup for `eventsContainer.cmp`.

```
<!--docsample:eventsContainer-->
<aura:component>
    <docsample:eventsHandler name="eventsHandler1"/>
    <docsample: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 Outside the Framework Lifecycle

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 146.

117

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

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

SEE ALSO:

Client-Side Rendering to the DOM

Events Best Practices

Events Fired During the Rendering Lifecycle

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

Component Creation

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

Let's look at an app with several nested components. The framework instantiates the app and goes through the children of the `v.body` facet to create each component, First, it creates the component definition, its entire parent hierarchy, and then creates the facets within those components. The framework also creates any component dependencies on the server, including definitions for attributes, interfaces, controllers, and actions..

The following image lists the order of component creation.

After creating a component instance, the serialized component definitions and instances are sent down to the client. Definitions are cached but not the instance data. The client deserializes the response to create the JavaScript objects or maps, resulting in an instance tree that's used to render the component instance. When the component tree is ready, the `init` event is fired for all the components, starting from the children component and finishing in the parent component.

Component Rendering

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

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

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

   You can customize the `init` handler and add your own controller logic before the component starts rendering. For more information, see Invoking Actions on Component Initialization on page 146.

2. For each component in the tree, the base implementation of `render()` or your custom renderer is called to start component rendering. For more information, see Client-Side Rendering to the DOM on page 136. Similar to the component creation process, rendering starts at the root component, its children components and their super components, if any, and finally the subchildren components.

3. Once your components are rendered to the DOM, `afterRender()` is called to signal that rendering is completed for each of these component definitions. It enables you to interact with the DOM tree after the framework rendering service has created the DOM elements.

4. To indicate that the client is done waiting for a response to the server request XHR, the `doneWaiting` event is fired. You can handle this event by adding a handler wired to a client-side controller action.

5. The framework checks whether any components need to be rerendered and rerenders any "dirtied" components to reflect any updates to attribute values. This rerender check is done even if there's no dirtied components or values.

6. Finally, the `doneRendering` event is fired at the end of the rendering lifecycle.

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

```
<!-- The uiExamples:buttonExample container component -->
<aura:component>
```

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

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

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

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

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

HTML tags in the markup are converted to `aura:html`, which has a `tag` attribute that defines the HTML tag to be generated. When rendering is done, this example calls `afterRender()` eight times for these component definitions. The `doneWaiting` event is fired, followed by the `doneRendering` event.

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

> **Note:** Firing an event in a custom renderer is not recommended. For more information, see Events Anti-Patterns.

Rendering Nested Components

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

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

SEE ALSO:

Client-Side Rendering to the DOM

System Event Reference

CHAPTER 8   Salesforce1 Events

Lightning components interact with Salesforce1 via events.

You can fire the following events, which are automatically handled by Salesforce1. If you fire these events in your Lightning apps or components outside of Salesforce1, you must handle them as necessary.

Event Name	Description
`force:createRecord`	Opens the page to create a new record for the specified `entityApiName`, for example, "Account" or "myNamespace__MyObject__c".
`force:editRecord`	Opens the page to edit the record specified by `recordId`.
`force:navigateToList`	Navigates to the list view specified by `listViewId`.
`force:navigateToObjectHome`	Navigates to the object home specified by the `scope` attribute.
`force:navigateToRelatedList`	Navigates to the related list specified by `parentRecordId`.
`force:navigateToSObject`	Navigates to an sObject record specified by `recordId`.
`force:navigateToURL`	Navigates to the specified URL.
`force:recordSave`	Saves a record.
`force:recordSaveSuccess`	Indicates that the record has been successfully saved.
`force:refreshView`	Reloads the view.
`force:showToast`	Displays a message in a popup.

Customizing Client-Side Logic for Salesforce1 and a Standalone App

Since Salesforce1 automatically handles many events, you have to do extra work if your component runs in a standalone app. Instantiating a Salesforce1 event using `$A.get()` can help you determine if your component is running within Salesforce1 or a standalone app. For example, you want to display a toast when a component loads in Salesforce1 and in a standalone app. You can fire the `force:showToast` event and set its parameters for Salesforce1 and create your own implementation for a standalone app.

```
displayToast : function (component, event, helper) {
    var toast = $A.get("e.force:showToast");
```

```
    if (toast){
        //fire the toast event in Salesforce1
        toast.setParams({
            "title": "Success!",
            "message": "The component loaded successfully."
        });
        toast.fire();
    } else {
        //your toast implementation for a standalone app here
    }
}
```

SEE ALSO:

Event Reference

CHAPTER 9   System Events

The framework fires several system events during its lifecycle.

You can handle these events in your Lightning apps or components, and within Salesforce1.

Event Name	Description
`aura:doneRendering`	Indicates that the initial rendering of the root application or root component has completed.
`aura:doneWaiting`	Indicates that the app or component is done waiting for a response to a server request. This event is preceded by an `aura:waiting` event.
`aura:locationChange`	Indicates that the hash part of the URL has changed.
`aura:noAccess`	Indicates that a requested resource is not accessible due to security constraints on that resource.
`aura:systemError`	Indicates that an error has occurred.
`aura:valueChange`	Indicates that a value has changed.
`aura:valueDestroy`	Indicates that a value is being destroyed.
`aura:valueInit`	Indicates that a value has been initialized.
`aura:waiting`	Indicates that the app or component is waiting for a response to a server request.

SEE ALSO:

System Event Reference

CREATING APPS

CHAPTER 10   App Basics

In this chapter ...

- App Overview
- Designing App UI
- Content Security
  Policy Overview

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 Using External JavaScript Libraries on page 131.

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.

CHAPTER 11 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`.

IN THIS SECTION:

Using External CSS

To reference an external CSS resource that you've uploaded as a static resource, use a `<ltng:require>` tag in your `.cmp` or `.app` markup.

SEE ALSO:

CSS in Components

Adding Lightning Components to Salesforce1

Using External CSS

To reference an external CSS resource that you've uploaded as a static resource, use a `<ltng:require>` tag in your `.cmp` or `.app` markup.

Here's an example of using `<ltng:require>`:

```
<ltng:require styles="/resource/resourceName" />
```

`resourceName` is the `Name` of the static resource. Note that the framework doesn't currently support the `$Resource` global variable available in Visualforce. In a managed packaged, the resource name must include the package namespace prefix, such as `/resource/np_resourceName`.

Here are some considerations for loading styles:

Loading Sets of CSS

Specify a comma-separated list of resources in the `styles` attribute to load a set of CSS.

Loading Order

The styles are loaded in the order that they are listed.

One-Time Loading

The styles only load once if they are specified in multiple `<ltng:require>` tags in the same component or across different components.

Encapsulation

To ensure encapsulation and reusability, add the `<ltng:require>` tag to every `.cmp` or `.app` resource that uses the CSS resource.

`<ltng:require>` also has a `scripts` attribute to load a list of JavaScript libraries. The `afterScriptsLoaded` event enables you to call a controller action after the `scripts` are loaded. It's only triggered by loading of the `scripts` and is never triggered when the CSS in `styles` is loaded.

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

Styling Components for Salesforce1

To prevent styling conflicts in Salesforce1, prefix your external CSS with a unique namespace. For example, if you prefix your external CSS declarations with `.myBootstrap`, wrap your component markup with a `<div>` tag that specifies the `myBootstrap` class.

```
<ltng:require styles="/resource/bootstrap"/>
<div class="myBootstrap">
    <c:myComponent />
    <!-- Other component markup -->
</div>
```

Note: Prefixing your CSS with a unique namespace only applies to external CSS. If you're using CSS within a component bundle, the `.THIS` keyword becomes `.namespaceComponentName` during runtime.

SEE ALSO:

Using External JavaScript Libraries

CSS in Components

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 12 Using JavaScript

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`.

`$A` is the shorthand in JavaScript code for the `Aura` object.

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. Modify the DOM only in `afterRender()` and `rerender()`. If you need to modify the DOM outside of the renderers, use utilities like `$A.util.addClass()`, `$A.util.removeClass()`, and `$A.util.toggleClass()`. Modify the DOM that belongs to the context component only.

Using Expressions

You can often avoid writing a custom renderer by using expressions in the markup instead. See Dynamically Showing or Hiding Markup on page 150 for more information.

Using External JavaScript Libraries

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

The framework's content security policy mandates that external JavaScript libraries must be uploaded to Salesforce static resources. For more information on static resources, see "What is a Static Resource?" in the Salesforce online help.

Here's an example of using `<ltng:require>`:

```
<ltng:require scripts="/resource/resourceName"
    afterScriptsLoaded="{!c.afterScriptsLoaded}" />
```

resourceName is the `Name` of the static resource. Note that the framework doesn't currently support the `$Resource` global variable available in Visualforce. In a managed packaged, the resource name must include the package namespace prefix, such as `/resource/np_resourceName`.

The `afterScriptsLoaded` action in the client-side controller is called after the scripts are loaded.

Here are some considerations for loading scripts:

Loading Sets of Scripts

Specify a comma-separated list of resources in the `scripts` attribute to load a set of resources.

Loading Order

The scripts are loaded in the order that they are listed.

One-Time Loading

Scripts only load once if they are specified in multiple `<ltng:require>` tags in the same component or across different components.

Parallel Loading

Use separate `<ltng:require>` tags for parallel loading if you have multiple sets of scripts that are not dependent on each other.

Encapsulation

To ensure encapsulation and reusability, add the `<ltng:require>` tag to every `.cmp` or `.app` resource that uses the JavaScript library.

`<ltng:require>` also has a `styles` attribute to load a list of CSS resources. You can set the `scripts` and `styles` attributes in one `<ltng:require>` tag.

If you're using an external library to work with your HTML elements after rendering, use `afterScriptsLoaded` to wire up a client-side controller. The following example sets up a chart using the `Chart.js` library, which is uploaded as a static resource.

```
<ltng:require scripts="/resource/chart"
              afterScriptsLoaded="{!c.setup}"/>
<canvas aura:id="chart" id="myChart" width="400" height="400"/>
```

The component's client-side controller sets up the chart after component initialization and rendering.

```
setup : function(component, event, helper) {
    var data = {
        labels: ["January", "February", "March"],
        datasets: [{
            data: [65, 59, 80, 81, 56, 55, 40]
            }]
    };
    var el = component.find("chart").getElement();
    var ctx = el.getContext("2d");
    var myNewChart = new Chart(ctx).Line(data);}
```

SEE ALSO:

Reference Doc App

Content Security Policy Overview

Using External CSS

Working with Attribute Values in JavaScript

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

`component.get(String key)` and `component.set(String key, Object value)` retrieves and assigns values associated with the specified key on the component. Keys are passed in as an expression, which represents attribute values. To retrieve an attribute value of a component reference, use `component.find("cmpId").get("v.value")`. Similarly, use `component.find("cmpId").set("v.value", myValue)` to set the attribute value of a component reference. This example shows how you can retrieve and set attribute values on a component reference, represented by the button with an ID of `button1`.

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

```
    <ui:button aura:id="button1" label="Button 1"/>
    {!v.buttonLabel}
    <ui:button label="Get Label" press="{!c.getLabel}"/>
</aura:component>
```

This controller action retrieves the `label` attribute value of a button in a component and sets its value on the `buttonLabel` attribute.

```
({
    getLabel : function(component, event, helper) {
        var myLabel = component.find("button1").get("v.label");
        component.set("v.buttonLabel", myLabel);
    }
})
```

In the following examples, `cmp` is a reference to a component in your JavaScript code.

Get an Attribute Value

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

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

Set an Attribute Value

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

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

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"));
```

Working with a Component Body in JavaScript

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

In these examples, `cmp` is a reference to a component in your JavaScript code. It's usually easy to get a reference to a component in JavaScript code. Remember that the `body` attribute is an array of components, so you can use the JavaScript `Array` methods on it.

> **Note:** When you use `cmp.set("v.body", ...)` to set the component body, you must explicitly include `{!v.body}` in your component markup.

Replace a Component's Body

To replace the current value of a component's body with another component:

```
// newCmp is a reference to another component
cmp.set("v.body", newCmp);
```

Clear a Component's Body

To clear or empty the current value of a component's body:

```
cmp.set("v.body", []);
```

Append a Component to a Component's Body

To append a `newCmp` component to a component's body:

```
var body = cmp.get("v.body");
// newCmp is a reference to another component
body.push(newCmp);
cmp.set("v.body", body);
```

Prepend a Component to a Component's Body

To prepend a `newCmp` component to a component's body:

```
var body = cmp.get("v.body");
body.unshift(newCmp);
cmp.set("v.body", body);
```

Remove a Component from a Component's Body

To remove an indexed entry from a component's body:

```
var body = cmp.get("v.body");
// Index (3) is zero-based so remove the fourth component in the body
body.splice(3, 1);
cmp.set("v.body", body);
```

SEE ALSO:

Component Body

Working with Attribute Values in JavaScript

Sharing JavaScript Code in a Component Bundle

Put functions that you want to reuse in the component's helper. Helper functions also enable specialization of tasks, such as processing data and firing server-side actions.

They can be called from any JavaScript code in a component's bundle, such as from a client-side controller or renderer. Helper functions are similar to client-side controller functions in shape, surrounded by brackets and curly braces to denote a JSON object containing a map of name-value pairs. A helper function can pass in any arguments required by the function, such as the component it belongs to, a callback, or any other objects.

Creating a Helper

A helper 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. You can also pass in an instance variable as a parameter, for example, `createExpense: function(component, expense){...}`, where `expense` is a variable defined in the component.

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

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

Helper functions are local to a component, improve code reuse, and move the heavy lifting of JavaScript logic away from the client-side controller where possible. The following code shows the helper function, which takes in the `value` parameter set in the controller via the `item` argument. The code walks through calling a server-side action and returning a callback but you can do something else in the helper function.

```
({
    updateItem : function(component, item, callback) {
        //Update the items via a server-side action
        var action = component.get("c.saveItem");
        action.setParams({"item" : item});
        //Set any optional callback and enqueue the action
        if (callback) {
            action.setCallback(this, callback);
        }
        $A.enqueueAction(action);
    }
})
```

SEE ALSO:

Client-Side Rendering to the DOM

Component Bundles

Handling Events with Client-Side Controllers

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.

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.

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.

Modifying Components Outside the Framework Lifecycle

Use `$A.run()` to wrap any code that modifies a component outside the normal rerendering lifecycle. The `$A.run()` call ensures that the framework rerenders the modified component and processes any enqueued actions.

You don't need to use `$A.run()` if your code is executed as part of the framework's call stack; for example, your code is handling an event or in the callback for a client-side controller action.

An example of where you need to use `$A.run()` is calling `window.setTimeout()` in an event handler to execute some logic after a time delay. This puts your code outside the framework's call stack.

This sample sets the `visible` attribute on a component to `true` after a five-second delay.

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

Note how the code updating a component attribute is wrapped in `$A.run()`, which ensures that the framework rerenders the modified component.

Note: Always add an `isValid()` check if you reference a component in asynchronous code, such as a timeout.

Validating Fields

You can validate fields using JavaScript. Typically, you validate the user input, identify any errors, and display the error messages.

Default Error Handling

The framework can handle and display errors using the default error component, `ui:inputDefaultError`. The following example shows how the framework handles a validation error and uses the default error component to display the error message.

Component source

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

Client-side controller source

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

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

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

Custom Error Handling

`ui:input` and its child components can handle errors using its `onError` and `onClearErrors` attributes, which are wired to your custom error handlers defined in a controller. `onError` maps to a `ui:validationError` event, and `onClearErrors` maps to `ui:clearErrors`. The input components can use the `ui:updateError` event to update the default error component, `ui:inputDefaultError`.

The following example shows how you can handle a validation error using custom error handlers and display the error message using the default error component.

Component source

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

Client-side controller source

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

        // is input numeric?
        if (isNaN(value)) {
            // fire event that will set error
            var errorEvent = inputCmp.getEvent("onError");
          errorEvent.setParams({ "errors" : [{message:"Input not a number: " + value}]});

            errorEvent.fire();
        } else {
            // fire event that will clear error
                var clearErrorEvent = inputCmp.getEvent("onClearErrors");
                clearErrorEvent.fire();
        }
    },

    handleError: function(component, event){
        var inputCmp = component.find("inputCmp");
        var errorsObj = event.getParam("errors");

        /* do any custom error handling
         * logic desired here */

        // set error using default error component
        inputCmp.setValid("v.value", false);
        inputCmp.addErrors(errorsObj);
        var updateErrorEvent = inputCmp.getEvent("updateError");
        updateErrorEvent.fire();
    },

    handleClearError: function(component, event) {
        var inputCmp = component.find("inputCmp");

        /* do any custom error handling
         * logic desired here */

        // clear error using default error component
        inputCmp.setValid("v.value", true);
        var updateErrorEvent = inputCmp.getEvent("updateError");
        updateErrorEvent.fire();
    }
}
```

When you enter a value and click **Submit**, an action in the controller executes. However, instead of letting the framework handle the errors, you have to provide a custom error handler using the `onError` attribute in the component. If the validation fails, `doAction` adds an error message using `setParams()` and fires your custom error handler. In the custom event handler, `handleError`, retrieve the errors by calling `getParam()` and invalidate the input value using `setValid(false)`. You can fire the `updateError` event to update the default error component.

Similarly, you can customize how you want to clear the errors by using the `onClearErrors` event. See the `handleClearError` handler in the controller for an example.

Throwing and Handling Errors

The framework gives you flexibility in handling unrecoverable and recoverable app errors in JavaScript code. For example, you can throw these errors when handling a server-side response using `action.setCallback()`.

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` to tell the user about the problem.

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

```
<!--docsample:recoverableError-->
<aura:component>
    <p>Click the button to trigger the controller to throw an error.</p>
    <div aura:id="div1"></div>

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

Here is the client-side controller source.

```
/*recoverableErrorController.js*/
({
    throwErrorForKicks: function(cmp) {
        // this sample always throws an error to demo try/catch
        var hasPerm = false;
        try {
            if (!hasPerm) {
                throw new Error("You don't have permission to edit this record.");
            }
        }
        catch (e) {
            $A.createComponents([
                ["ui:message",{
                    "title" : "Sample Thrown Error",
                    "severity" : "error",
                }],
                ["ui:outputText",{
                    "value" : e.message
```

```
                }]
                ],
                function(components, status){
                    if (status === "SUCCESS") {
                        var message = components[0];
                        var outputText = components[1];
                        // set the body of the ui:message to be the ui:outputText
                        message.set("v.body", outputText);
                        var div1 = cmp.find("div1");
                        // Replace div body with the dynamic component
                        div1.set("v.body", message);
                    }
                }
            );
        }
    }
})
```

The controller code throws an error and catches it. The message in the error is displayed to the user in a dynamically created `ui:message` component. The body of the `ui:message` is a `ui:outputText` component containing the error text.

SEE ALSO:

Validating Fields

Dynamically Creating Components

Making API Calls

You can't make API calls from client-side code. Make API calls, including Salesforce API calls, from server-side controllers instead.

The framework uses Content Security Policy (CSP) to control the source of content that can be loaded on a page. Lightning apps are served from a different domain than Salesforce APIs so the CSP doesn't allow API calls from JavaScript code.

For information about making API calls from server-side controllers, see

SEE ALSO:

Content Security Policy Overview

CHAPTER 13   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}"/>

    <p>This value is set in the controller after the component initializes and before
rendering.</p>
    <p><b>{!v.setMeOnInit}</b></p>

</aura:component>
```

Client-side controller source

```
({
    doInit: function(cmp) {
        // Set the attribute value.
        // You could also fire an event here instead.
        cmp.set("v.setMeOnInit", "controller init magic!");
    }
})
```

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

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

This registers an `init` event handler for the component. `init` is a predefined event sent to every component. After the component is initialized, the `doInit` action is called in the component's controller. In this sample, the controller action sets an attribute value, but it could do something more interesting, such as firing an event.

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 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 more information, see aura:valueChange on page 284.

SEE ALSO:

Invoking Actions on Component Initialization

Finding Components by ID

Retrieve a component by its ID in JavaScript code. For example, you can add a local ID of `button1` to the `ui:button` component.

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

You can find the component by calling `cmp.find("button1")`, where `cmp` is a reference to the component containing the button. The `find()` function has one parameter, which is the local ID of a component within the markup.

You can also retrieve a component by its global ID, which is an ID generated during runtime.

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

For example, the `ui:button` component renders as an HTML button element with this markup.

```
<button class="default uiButton" data-aura-rendered-by="30:463;a">...</button>
```

Retrieve the component by using `$A.getCmp("30:463;a")`.

SEE ALSO:

Component IDs

Value Providers

Dynamically Creating Components

Create a component dynamically in your client-side JavaScript code by using the `$A.createComponent()` method.

📝 **Note:** Use `createComponent()` instead of the deprecated `newComponent()`, `newComponentAsync()`, and `newComponentDeprecated()` methods.

The syntax is:

```
createComponent(String type, Object attributes, function callback)
```

1. `type`—The type of component to create; for example, `"ui:button"`.

2. `attributes`—A map of attributes for the component.

3. `callback`—The callback to invoke after the component is created. The new component is passed in to the callback as a parameter.

Let's add a dynamically created button to this sample component.

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

    <p>Dynamically created button</p>
    {!v.body}

</aura:component>
```

The client-side controller calls `$A.createComponent()` to create the button with a local ID and a handler for the `press` event. The button is appended to the `body` of `docsample:createComponent`.

```
/*createComponentController.js*/
({
    doInit : function(cmp) {
        $A.createComponent(
            "ui:button",
            {
                "aura:Id": "findableAuraId",
                "label": "Press Me",
                "press": cmp.getReference("c.handlePress")
            },
            function(newButton){
                //Add the new button to the body array
                if (cmp.isValid()) {
                    var body = cmp.get("v.body");
                    body.push(newButton);
                    cmp.set("v.body", body);
                }
            }
        );
    },

    handlePress : function(cmp) {
        console.log("button pressed");
    }
})
```

📝 **Note:** `docsample:createComponent` contains a `{!v.body}` expression. When you use `cmp.set("v.body", ...)` to set the component body, you must explicitly include `{!v.body}` in your component markup.

To retrieve the new button you created, use `body[0]`.

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

Creating Nested Components

To dynamically create a component in the body of another component, use `$A.createComponents()` to create the components. In the function callback, nest the components by setting the inner component in the `body` of the outer component. This example creates a `ui:outputText` component in the `body` of a `ui:message` component.

```
$A.createComponents([
    ["ui:message",{
        "title" : "Sample Thrown Error",
        "severity" : "error",
    }],
    ["ui:outputText",{
        "value" : e.message
    }]
    ],
    function(components, status){
        if (status === "SUCCESS") {
            var message = components[0];
            var outputText = components[1];
            // set the body of the ui:message to be the ui:outputText
            message.set("v.body", outputText);
        }
    }
);
```

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.

Server-Side Dependencies

The `createComponent()` 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:

> Reference Doc App
>
> aura:dependency
>
> Invoking Actions on Component Initialization
>
> 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 `docsample:sampleComponent`.

```
addNewHandler : function(cmp, event) {
    var cmpArr = cmp.find({ instancesOf : "docsample: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.createComponent()`. For more information, see Dynamically Creating Components.

`addHandler()` adds an event handler to a component.

Note that you can't 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

Dynamically Showing or Hiding Markup

Use CSS to toggle markup visibility. You could use the `<aura:if>` or `<aura:renderIf>` tags to do the same thing but we recommend using CSS as it's the more standard approach.

This example uses `$A.util.toggleClass(element, 'class')` to toggle visibility of markup.

```
<!--docsample:toggleCss-->
<aura:component>
```

```
    <ui:button label="Toggle" press="{!c.toggle}"/>
    <p aura:id="text">Now you see me</p>
</aura:component>
```

```
/*toggleCssController.js*/
({
    toggle : function(component, event, helper) {
        var toggleText = component.find("text");
        $A.util.toggleClass(toggleText, "toggle");
    }
})
```

```
/*toggleCss.css*/
.THIS.toggle {
    display: none;
}
```

Click the **Toggle** button to hide or show the text by toggling the CSS class.

SEE ALSO:

Handling Events with Client-Side Controllers

Component Attributes

Adding and Removing Styles

Adding and Removing Styles

You can add or remove a CSS style 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.

Toggling a Class

To toggle a class, use `$A.util.toggleClass(element, 'class')`, which adds or removes the class. The `element` parameter can be an HTML element or a component.

To hide or show markup dynamically, see Dynamically Showing or Hiding Markup on page 150.

To conditionally set a class for an array of elements, pass in the array to `$A.util.toggleClass()`.

```
mapClasses: function(arr, cssClass) {
    for(var element in arr) {
        $A.util.toggleClass(arr[element], cssClass);
    }
}
```

📝 Note: If the utility function is not used inside `afterRender()` or `rerender()`, passing in `cmp.getElement()` might result in your class not being applied when the components are rerendered. In such cases, passing in the component `cmp` is recommended over passing in the HTML element. For more information, see Events Fired During the Rendering Lifecycle on page 118.

Refer to the JavaScript API Reference for more utility functions for working with DOM elements.

SEE ALSO:

Handling Events with Client-Side Controllers

CSS in Components

Component Bundles

CHAPTER 14   Using Apex

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.

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 155.

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.

Creating an Apex Server-Side Controller

Use the Developer Console to create an Apex server-side controller.

Calling a Server-Side Action

Call a server-side controller action from a client-side controller. In the client-side controller, you set a callback, which is called after the server-side action is completed. A server-side action can return any object containing serializable JSON data.

Queueing of Server-Side Actions

The framework queues up actions before sending them to the server. This mechanism is largely transparent to you when you're writing code but it enables the framework to minimize network traffic by batching multiple actions into one request.

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. We recommend that you only use abortable actions for read-only operations as they are not guaranteed to be sent to the server.

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.

Only methods that you have explicitly annotated with `@AuraEnabled` are exposed.

Tip: Don't store component state in your controller. Store it in a component's attribute instead.

This Apex controller contains a `serverEcho` action that prepends a string to the value passed in.

```
public with sharing class SimpleServerSideController {

    //Use @AuraEnabled to enable client- and server-side access to the method
    @AuraEnabled
    public static String serverEcho(String firstName) {
        return ('Hello from the server, ' + firstName);
```

```
    }
}
```

Creating an Apex Server-Side Controller

Use the Developer Console to create an Apex server-side controller.

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.

   📝 **Note:** Add the `@AuraEnabled` annotation to any methods, including getters and setters, that you wish to expose on the client- or server-side. This means that you only expose methods that you have explicitly annotated.

6. Click **File** > **Save**.

7. Open the component that you want to wire to the new controller class.

8. Add a `controller` system attribute to the `<aura:component>` tag to wire the component to the controller. For example:

```
<aura:component controller="SimpleServerSideController" >
```

Calling a Server-Side Action

Call a server-side controller action from a client-side controller. In the client-side controller, you set a callback, which is called after the server-side action is completed. A server-side action can return any object containing serializable JSON data.

A client-side controller is a JavaScript object in object-literal notation containing name-value pairs. Each name corresponds to a client-side action. Its value is the function code associated with the action.

Let's say that you want to trigger a server-call from a component. The following component contains a button that's wired to a client-side controller `echo` action. `SimpleServerSideController` contains a method that returns a string passed in from the client-side controller.

```
<aura:component controller="SimpleServerSideController">
    <aura:attribute name="firstName" type="String" default="world"/>
    <ui:button label="Call server" press="{!c.echo}"/>
</aura:component>
```

The following client-side controller includes an `echo` action that executes a `serverEcho` method 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. `action.setParams({ firstName : cmp.get("v.firstName")`

155

}); retrieves the `firstName` attribute from the component and sets the value of the `firstName` argument on the server-side controller's `serverEcho` method.

```
({
    "echo" : function(cmp) {
        // create a one-time use instance of the serverEcho action
        // in the server-side controller
        var action = cmp.get("c.serverEcho");
        action.setParams({ firstName : cmp.get("v.firstName") });

        // Create a callback that is executed after
        // the server-side action returns
        action.setCallback(this, function(response) {
            var state = response.getState();
            // This callback doesn't reference cmp. If it did,
            // you should run an isValid() check
            //if (cmp.isValid() && state === "SUCCESS") {
            if (state === "SUCCESS") {
                // Alert the user with the value returned
                // from the server
                alert("From server: " + response.getReturnValue());

                // You would typically fire a event here to trigger
                // client-side notification that the server-side
                // action is complete
            }
            //else if (cmp.isValid() && state === "ERROR") {
            else if (state === "ERROR") {
                var errors = response.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");
                }
            }
        });

        // optionally set abortable flag here

        // A client-side action could cause multiple events,
        // which could trigger other events and
        // other server-side action calls.
        // $A.enqueueAction adds the server-side action to the queue.
        $A.enqueueAction(action);
    }
})
```

In the client-side controller, we use the value provider of `c` to invoke a server-side controller action. 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.

> ✏️ **Note:** Always add an `isValid()` check if you reference a component in asynchronous code, such as a callback.

Action States

The possible action states are:

NEW
> The action was created but is not in progress yet

RUNNING
> The action is in progress

SUCCESS
> The action executed successfully

ERROR
> The server returned an error

ABORTED
> The action was aborted

> ✏️ **Note:** `setCallback()` has a third parameter that registers the action state that will invoke the callback. If you don't specify the third argument for `setCallback()`, it defaults to registering the `SUCCESS` and `ERROR` states. To set a callback for another state, such as `ABORTED`, you can call `setCallback()` multiple times with the action state set explicitly in the third argument. For example:

```
action.setCallback(this, function(response) { ...}, "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 by batching multiple actions into one request.

Event processing can generate a tree of events if an event handler fires more events. The framework processes the event tree and adds every action that needs to be executed on the server to a queue.

When the tree of events and all the client-side actions are processed, the framework batches actions from the queue into a message before sending it to the server. A message is essentially a wrapper around a list of actions.

> 💡 **Tip:** If your action is not executing, make sure that you're not executing code outside the framework's normal rerendering lifecycle. For example, if you use `window.setTimeout()` in an event handler to execute some logic after a time delay, you must wrap your code in `$A.run()`.

SEE ALSO:

   Modifying Components Outside the Framework Lifecycle

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. We recommend that you only use abortable actions for read-only operations as they are not guaranteed to be sent to the server.

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 navigation item, all abortable actions are removed from the queue. The aborted actions are not sent to the server and their state is set to `ABORTED`.

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.

If some actions have been sent but not yet returned from the server, they will complete, but only the callback logic associated with the `ABORTED` state (`action.getState() === "ABORTED"`) will be executed. This enables components to optionally log a message or clean up if they had an aborted action.

> 📝 **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.

Marking an Action as Abortable

Mark a server-side action as abortable by using the `setAbortable()` method on the `Action` object in JavaScript. For example:

```
var action = cmp.get("c.serverEcho");
action.setAbortable();
```

`setCallback()` has a third parameter that registers the action state that will invoke the callback. If you don't specify the third argument for `setCallback()`, it defaults to registering the `SUCCESS` and `ERROR` states.To check for aborted actions in your callback and take appropriate action, such as logging the aborted action, call `setCallback()` with the `ABORTED` state set explicitly in the third argument. For example:

```
// Process default action states
action.setCallback(this, function(response) {
    var state = response.getState();
    if (state === "SUCCESS") {
        // Alert the user with the value returned from the server
        alert("From server: " + response.getReturnValue());
    }
    // process other action states
});
// Explicitly register callback for ABORTED
action.setCallback(this,
    function(response) {
        alert("The action was aborted");
```

```
    },
    "ABORTED"
);
```

Rapid Clicking

Imagine a navigation menu where each action is a potentially slow request to the server. A user may click on several navigation items quickly so that none of the server responses return before the subsequent click. If all the actions are marked as abortable, none of the callbacks will be called except for the last click. This improves user experience by avoiding flickering due to sequential rendering of multiple server responses.

Progressive Loading

Sometimes, you might want to do a progressive loading of data where the first set of items is loaded, followed by subsequent data loads after the rendering of the first set is complete. You can do this by calling a second set of actions after a delay, and using the `setParentAction()` method in the `Action` object to associate each action in the second set with one of the actions in the first set. This ensures that the second set of actions will abort if the user navigates away.

SEE ALSO:

> Creating Server-Side Logic with Controllers
>
> Queueing of Server-Side Actions
>
> Calling a Server-Side Action

Creating Components

The `Cmp.<myNamespace>.<myComponent>` syntax to reference a component in Apex is deprecated. Use `$A.createComponent()` in client-side JavaScript code instead.

SEE ALSO:

> Dynamically Creating Components

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 with sharing class AccountController {

    @AuraEnabled
    public static void updateAnnualRevenue(String accountId, Decimal annualRevenue) {
        Account acct = [SELECT Id, Name, BillingCity FROM Account WHERE Id = :accountId];

        acct.AnnualRevenue = annualRevenue;

        // Perform isAccessible() and isUpdateable() checks here
        update acct;
    }
}
```

For an example of calling Apex code from JavaScript code, see the Quick Start on page 6.

Loading Record Data from a Standard Object

Load records from a standard object in a server-side controller. The following server-side controller has methods that return a list of opportunity records and an individual opportunity record.

```
public with sharing 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
         ];

        // Perform isAccessible() check here
        return opportunity;
    }
}
```

This example component uses the previous server-side controller to display a list of opportunity records when you press a button.

```
<aura:component controller="OpportunityController">
    <aura:attribute name="opportunities" type="Opportunity[]"/>

    <ui:button label="Get Opportunities" press="{!c.getOpps}"/>
    <aura:iteration var="opportunity" items="{!v.opportunities}">
     <p>{!opportunity.Name} : {!opportunity.CloseDate}</p>
```

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

When you press the button, the following client-side controller calls the `getOpportunities()` server-side controller and sets the `opportunities` attribute on the component. For more information about calling server-side controller methods, see Calling a Server-Side Action on page 155.

```
({
    getOpps: function(cmp){
        var action = cmp.get("c.getOpportunities");
        action.setCallback(this, function(response){
            var state = response.getState();
            if (state === "SUCCESS") {
                cmp.set("v.opportunities", response.getReturnValue());
            }
        });
    $A.enqueueAction(action);
    }
})
```

> 📝 **Note:** To load record data during component initialization, use the `init` handler.

Loading Record Data from a Custom Object

Load record data using an Apex controller and setting the data on a component attribute. This server-side controller returns records on a custom object `myObj__c`.

```
public with sharing class MyObjController {

    @AuraEnabled
    public static List<MyObj__c> getMyObjects() {

        // Perform isAccessible() checks here
        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="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. This step can also be done during component initialization using the `init` handler.

```
getMyObjects: function(cmp){
    var action = cmp.get("c.getMyObjects");
    action.setCallback(this, function(response){
        var state = response.getState();
        if (state === "SUCCESS") {
            cmp.set("v.myObjects", response.getReturnValue());
```

```
        }
    });
    $A.enqueueAction(action);
}
```

For an example on loading and updating records using controllers, see the Quick Start on page 6.

IN THIS SECTION:

CRUD and Field-Level Security (FLS)

In addition to the Content Security Policy, Lightning Components imposes CRUD and field-level security to ensure component security.

Saving Records

You can take advantage of the built-in create and edit record pages in Salesforce1 to create or edit records via a Lightning component.

Deleting Records

You can delete records via a Lightning component to remove them from both the view and database.

SEE ALSO:

CRUD and Field-Level Security (FLS)

CRUD and Field-Level Security (FLS)

In addition to the Content Security Policy, Lightning Components imposes CRUD and field-level security to ensure component security.

Lightning components don't automatically enforce CRUD and FLS when you reference objects or retrieve the objects from an Apex controller. This means that the framework continues to display records and fields for which users don't have CRUD access and FLS visibility. You must manually reinforce CRUD and FLS in your Apex controllers. For example, including the `with sharing` keyword in an Apex controller ensures that users see only the records they have access to in a Lightning component. Additionally, you must explicitly check for `isAccessible()`, `isCreateable()`, `isDeletable()`, and `isUpdateable()` prior to performing operations on records or objects.

This example shows the recommended way to perform an operation on the expense object introduced in Create a Standalone Lightning App on page 8.

```
public with sharing class ExpenseController {

    // ns refers to namespace; leave out ns__ if not needed
    // This method is vulnerable.
    @AuraEnabled
    public static List<ns__Expense__c> get_UNSAFE_Expenses() {
        return [SELECT Id, Name, ns__Amount__c, ns__Client__c, ns__Date__c,
            ns__Reimbursed__c, CreatedDate FROM ns__Expense__c];
     }

    // This method is recommended.
    @AuraEnabled
    public static List<ns__Expense__c> getExpenses() {
        String [] expenseAccessFields = new String [] {'Id',
                                                        'Name',
                                                        'ns__Amount__c',
                                                        'ns__Client__c',
```

```
                                                'ns__Date__c',
                                                'ns__Reimbursed__c',
                                                'CreatedDate'
                                                };


    // Obtain the field name/token map for the Expense object
    Map<String,Schema.SObjectField> m = Schema.SObjectType.ns__Expense__c.fields.getMap();


    for (String fieldToCheck : expenseAccessFields) {

        // Check if the user has access to view field
        if (!m.get(fieldToCheck).getDescribe().isAccessible()) {

            // Pass error to client
            throw new System.NoAccessException()

            // Suppress editor logs
            return null;
        }
    }

    // Query the object safely
    return [SELECT Id, Name, ns__Amount__c, ns__Client__c, ns__Date__c,
            ns__Reimbursed__c, CreatedDate FROM ns__Expense__c];
    }
}
```

📝 Note: For more information, see the articles on Enforcing CRUD and FLS and Lightning Security.

Saving Records

You can take advantage of the built-in create and edit record pages in Salesforce1 to create or edit records via a Lightning component.

The following component contains a button that calls a client-side controller to display the edit record page.

```
<aura:component>
    <ui:button label="Edit Record" press="{!c.edit}"/>
</aura:component>
```

The client-side controller fires the `force:recordEdit` event, which displays the edit record page for a given contact ID. For this event to be handled correctly, the component must be included in Salesforce1.

```
edit : function(component, event, helper) {
    var editRecordEvent = $A.get("e.force:editRecord");
    editRecordEvent.setParams({
        "recordId": component.get("v.contact.Id")
    });
    editRecordEvent.fire();
}
```

Records updated using the `force:recordEdit` event are persisted by default.

Saving Records using a Lightning Component

Alternatively, you might have a Lightning component that provides a custom form for users to add a record. To save the new record, wire up a client-side controller to an Apex controller. The following list shows how you can persist a record via a component and Apex controller.

📝 **Note:** If you create a custom form to handle record updates, you must provide your own field validation.

Create an Apex controller to save your updates with the `upsert` operation. The following example is an Apex controller for upserting record data.

```
@AuraEnabled
public static Expense__c saveExpense(Expense__c expense) {
    // Perform isUpdateable() check here
    upsert expense;
    return expense;
}
```

Call a client-side controller from your component. For example, `<ui:button label="Submit" press="{!c.createExpense}"/>`.

In your client-side controller, provide any field validation and pass the record data to a helper function.

```
createExpense : function(component, event, helper) {
    // Validate form fields
    // Pass form data to a helper function
    var newExpense = component.get("v.newExpense");
    helper.createExpense(component, newExpense);
}
```

In your component helper, get an instance of the server-side controller and set a callback. The following example upserts a record on a custom object. Recall that `setParams()` sets the value of the `expense` argument on the server-side controller's `saveExpense()` method.

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

SEE ALSO:

   CRUD and Field-Level Security (FLS)

Deleting Records

You can delete records via a Lightning component to remove them from both the view and database.

Create an Apex controller to delete a specified record with the `delete` operation. The following Apex controller deletes an expense object record.

```
@AuraEnabled
public static Expense__c deleteExpense(Expense__c expense) {
    // Perform isDeletable() check here
    delete expense;
    return expense;
}
```

Depending on how your components are set up, you might need to create an event to tell another component that a record has been deleted. For example, you have a component that contains a sub-component that is iterated over to display the records. Your sub-component contains a button (1), which when pressed fires an event that's handled by the container component (2), which deletes the record that's clicked on.

```
<aura:registerEvent name="deleteExpenseItem" type="c:deleteExpenseItem"/>
<ui:button label="Delete" press="{!c.delete}"/>
```

Create a component event to capture and pass the record that's to be deleted. Name the event `deleteExpenseItem`.

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

Then, pass in the record to be deleted and fire the event in your client-side controller.

```
delete : function(component, evt, helper) {
    var expense = component.get("v.expense");
    var deleteEvent = component.getEvent("deleteExpenseItem");
    deleteEvent.setParams({ "expense": expense }).fire();
}
```

In the container component, include a handler on your sub-component. In this example, `c:expenseList` is the sub-component.

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

And handle the event in the client-side controller of the container component.

```
deleteEvent : function(component, event, helper) {
    // Call the helper function to delete record and update view
    helper.deleteExpense(component, event.getParam("expense"));
}
```

Finally, in the helper function of the container component, call your Apex controller to delete the record and update the view.

```
deleteExpense : function(component, expense, callback) {
    // Call the Apex controller and update the view in the callback
    var action = component.get("c.deleteExpense");
    var self = this;
    action.setParams({
        "expense": expense
    });
    action.setCallback(this, function(response) {
        var state = response.getState();
        if (state === "SUCCESS") {
            // Remove only the deleted expense from view
            var expenses = component.get("v.expenses");
            var items = [];
            for (i = 0; i < expenses.length; i++) {
                if(expenses[i]!==expense) {
                    items.push(expenses[i]);
                }
            }
            component.set("v.expenses", items);
            // Other client-side logic
        }
    });
    $A.enqueueAction(action);
}
```

The helper function calls the Apex controller to delete the record in the database. In the callback function, `component.set("v.expenses", items)` updates the view with the updated array of records.

SEE ALSO:

CRUD and Field-Level Security (FLS)

Create a Standalone Lightning App

Component Events

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 Apex to a production organization, each unit test in your organization namespace is executed by default.

  - 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 8.

```apex
@isTest
class TestExpenseController {
    static testMethod void test() {
        //Create new expense and insert it into the database
        Expense__c exp = new Expense__c(name='My New Expense',
                            amount__c=20, client__c='ABC',
                            reimbursed__c=false, date__c=null);
         ExpenseController.saveExpense(exp);

        //Assert the name field and saved expense
        System.assertEquals('My New Expense',
                        ExpenseController.getExpenses()[0].Name,
                        'Name does not match');
        System.assertEquals(exp, ExpenseController.saveExpense(exp));
    }
}
```

Note: Apex classes must be manually added to your package.

For more information on distributing Apex code, see the *Apex Code Developer's Guide*.

SEE ALSO:

Distributing Applications and Components

Making API Calls from Apex

Make API calls from an Apex controller. You can't make API calls from JavaScript code.

For information about making API calls from Apex, see the *Force.com Apex Code Developer's Guide*.

CHAPTER 15 Using Object-Oriented Development

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

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

You can extend a component, app, or interface, or you can implement a component interface.

169

What is Inherited?

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

Component Attributes

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

Events

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

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

Helpers

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

Controllers

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

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

SEE ALSO:

   Component Attributes

   Events

   Sharing JavaScript Code in a Component Bundle

   Handling Events with Client-Side Controllers

   aura:set

Inherited Component Attributes

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

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

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

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

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

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

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

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

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

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

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

Inherited **body** Attribute

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

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

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

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

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

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

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

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

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

`docsample:subBody` outputs:

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

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

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

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

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

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

SEE ALSO:

aura:set

Component Body

Component Markup

Abstract Components

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

Similarly, the Lightning Component framework supports the concept of abstract components that have a partial implementation but leave the remaining implementation to concrete sub-components.

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

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

SEE ALSO:

Interfaces

Interfaces

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 these tags:

- `<aura:attribute>` tags to define the interface's attributes.
- `<aura:registerEvent>` tags to define the events that it may fire.

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

To use an interface, you must implement it. An interface can't be used directly in markup otherwise. Set the `implements` system attribute in the `<aura:component>` tag to the name of the interface that you are implementing. For example:

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

A component can implement an interface and extend another component.

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

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

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

> 📝 Note: Use `<aura:set>` in a sub component to set the value of any attribute that is inherited from the super component. This usage works for components and abstract components, but it doesn't work for interfaces. To set the value of an attribute inherited from an interface, redefine the attribute in the sub component using `<aura:attribute>` and set the value in its default attribute.

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

SEE ALSO:

Setting Attributes Inherited from an Interface

Abstract Components

Marker Interfaces

You can use an interface as a marker interface that is implemented by a set of components that you want to easily identify for specific usage in your app.

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

Inheritance Rules

This table describes the inheritance rules for various elements.

Element	extends	implements	Default Base Element
component	one extensible component	multiple interfaces	`<aura:component>`
app	one extensible app	N/A	`<aura:application>`

173

Element	extends	implements	Default Base Element
interface	multiple interfaces using a comma-separated list (extends="ns:intf1,ns:int2")	N/A	N/A

SEE ALSO:

[Interfaces](#)

CHAPTER 16   Using the AppCache

Application cache (AppCache) speeds up app response time and reduces server load by only downloading resources that have changed. It improves page loads affected by limited browser cache persistence on some devices.

AppCache can be useful if you're developing apps for mobile devices, which sometimes have very limited browser cache. Apps built for desktop clients may not benefit from the AppCache. The framework supports AppCache for WebKit-based browsers, such as Chrome and Safari.

Note: See an introduction to AppCache for more information.

SEE ALSO:

aura:application

175

Enabling the AppCache

The framework disables the use of AppCache by default.

To enable AppCache in your application, set the `useAppcache="true"` system attribute in the `aura:application` tag. We recommend disabling AppCache during initial development while your app's resources are still changing. Enable AppCache when you are finished developing the app and before you start using it in production to see whether AppCache improves the app's response time.

Loading Resources with AppCache

A cache manifest file is a simple text file that defines the Web resources to be cached offline in the AppCache.

The cache manifest is auto-generated for you at runtime if you have enabled AppCache in your application. If there are any changes to the resources, the framework updates the timestamp to trigger a refetch of all resources. Fetching resources only when necessary reduces server trips for users.

When a browser initially requests an app, a link to the manifest file is included in the response.

```
<html manifest="/path/to/app.manifest">
```

The manifest path includes the mode and app name of the app that's currently running. This manifest file lists framework resources as well as your JavaScript code and CSS, which are cached after they're downloaded for the first time. A hash in the URL ensures that you always have the latest resources.

> **Note:** You'll see different resources depending on which mode you're running in. For example, `aura_prod.js` is available in `PROD` mode and `aura_proddebug.js` is available in `PRODDEBUG` mode.

CHAPTER 17   Controlling Access

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.

Setting `access="global"` makes a bundle available to package subscribers and other namespaces. For more information on packaging, see Distributing Applications and Components on page 180

177

Application Access Control

The `access` attribute on the `aura:application` tag indicates whether the app can be used outside of the app's namespace.
Possible values are listed below.

Modifier	Description
`global`	The app can be used in any namespace.
`public`	The app can be used 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 18   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.

A managed package ensures that your application and other resources are fully upgradeable. To create and work with managed packages, you must use a Developer Edition organization and register a namespace prefix. 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

DEBUGGING

CHAPTER 19   Debugging

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 Google Chrome's 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**.

Log Messages

To help debug your client-side code, you can write output to the JavaScript console of a web browser.

Use the `$A.log(string, [error])` method to output a log message to the JavaScript console.

The first parameter is the string to log.

The optional second parameter is an error object that can include more detail.

> 📝 Note:
> - `$A` is the shorthand in JavaScript code for the `Aura` object.
> - `$A.log()` doesn't output in `PROD` or `PRODDEBUG` modes.

For example, `$A.log("This is a log message")` outputs to the JavaScript console:

```
This is a log message
```

If you put `$A.log("The name of the action is: " + this.getDef().getName())` inside an action called `openNote` in a client-side controller, it outputs to the JavaScript console:

```
The name of the action is: openNote
```

For instructions on using the JavaScript console, refer to the instructions for your web browser.

Logging in Production Modes

To log messages in `PROD` or `PRODDEBUG` modes, you can write a custom logging function. You must use `$A.logger.subscribe(String level, function callback)` to subscribe to log messages at a certain severity level.

The first parameter is the severity level you're subscribing to. The valid values are:

- `ASSERT`
- `ERROR`
- `INFO`
- `WARNING`

The second parameter is the callback function that will be called when a message at the subscribed severity level is logged. Note that `$A.log()` logs a message at the `INFO` severity level.

Let's look at some sample JavaScript code in a client-side controller.

```
({
    sampleControllerAction: function(cmp) {
        // subscribe to severity levels
        $A.logger.subscribe("INFO", logCustom);
        // Following subscriptions not exercised here but shown for completeness
        //$A.logger.subscribe("WARNING", logCustom);
        //$A.logger.subscribe("ASSERT", logCustom);
        //$A.logger.subscribe("ERROR", logCustom);

        $A.log("log one arg");
        $A.log("log two args", {message: "drat and double drat"});

        function logCustom(level, message, error) {
            console.log(getTimestamp(), "logCustom: ", arguments);
        }

        function getTimestamp() {
            return new Date().toJSON();
        }
    }

})
```

`$A.logger.subscribe("INFO", logCustom)` subscribes so that messages logged at the `INFO` severity level will call the `logCustom()` function. In this case, `logCustom()` simply logs the message to the console with a timestamp.

The `$A.log()` calls log messages at the `INFO` severity level, which matches the subscription and invokes the `logCustom()` callback.

Warning Messages

To help debug your client-side code, you can use the `warning()` method to write output to the JavaScript console of your web browser.

Use the `$A.warning(string)` method to write a warning message to the JavaScript console. The parameter is the message to display. For example, `$A.warning("This is a warning message.");` 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.

REFERENCE

CHAPTER 20 Reference Overview

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:dependency

The `<aura:dependency>` tag enables you to declare dependencies that can't easily be discovered by the framework.

The framework automatically tracks dependencies between definitions, such as components. This enables the framework to automatically reload when it detects that you've changed a definition during development. However, if a component uses a client- or server-side provider that instantiates components that are not directly referenced in the component's markup, use `<aura:dependency>` in the component's markup to explicitly tell the framework about the dependency. Adding the `<aura:dependency>` tag ensures that a component and its dependencies are sent to the client, when needed.

For example, adding this tag to a component marks the `aura:placeholder` component as a dependency.

```
<aura:dependency resource="markup://aura:placeholder" />
```

The `<aura:dependency>` tag includes these system attributes.

System Attribute	Description
resource	The resource that the component depends on. For example, `resource="markup://sampleNamespace:sampleComponent"` refers to the `sampleComponent` in the `sampleNamespace` namespace.
	Use an asterisk (*) in the resource name for wildcard matching. For example, `resource="markup://sampleNamespace:*"` matches everything in the namespace; `resource="markup://sampleNamespace:input*"` matches everything in the namespace that starts with `input`.
	Don't use an asterisk (*) in the namespace portion of the resource name. For example, `resource="markup://sample*:sampleComponent"` is not supported.
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.

Attribute	Type	Description
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: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:

Interfaces

Interface Access Control

aura:set

Use `<aura:set>` in markup to set the value of an attribute inherited from a super component, event, or interface.

To learn more, see:

- Setting Attributes Inherited from a Super Component
- Setting Attributes on a Component Reference
- Setting Attributes Inherited from an Interface

Setting Attributes Inherited from a Super Component

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

Let's look at an example. Here is the `docsample:setTagSuper` component.

```
<!--docsample:setTagSuper-->
<aura:component extensible="true">
```

```
    <aura:attribute name="address1" type="String" />
    setTagSuper address1: {!v.address1}<br/>
</aura:component>
```

`docsample:setTagSuper` outputs:

```
setTagSuper address1:
```

The `address1` attribute doesn't output any value yet as it hasn't been set.

Here is the `docsample:setTagSub` component that extends `docsample:setTagSuper`.

```
<!--docsample:setTagSub-->
<aura:component extends="docsample:setTagSuper">
    <aura:set attribute="address1" value="808 State St" />
</aura:component>
```

`docsample:setTagSub` outputs:

```
setTagSuper address1: 808 State St
```

`sampleSetTagExdocsample:setTagSub` sets a value for the `address1` attribute inherited from the super component, `docsample:setTagSuper`.

⚠️ **Warning:** This usage of `<aura:set>` works for components and abstract components, but it doesn't work for interfaces. For more information, see Setting Attributes Inherited from an Interface on page 189.

If you're using a component by making a reference to it in your component, you can set the attribute value directly in the markup. For example, `docsample:setTagSuperRef` makes a reference to `docsample:setTagSuper` and sets the `address1` attribute directly without using `aura:set`.

```
<!--docsample:setTagSuperRef-->
<aura:component>
    <docsample:setTagSuper address1="1 Sesame St" />
</aura:component>
```

`docsample:setTagSuperRef` outputs:

```
setTagSuper address1: 1 Sesame St
```

SEE ALSO:

Component Body

Inherited Component Attributes

Setting Attributes on a Component Reference

Setting Attributes on a Component Reference

When you include another component, such as `<ui:button>`, in a component, we call that a component reference to `<ui:button>`. You can use `<aura:set>` to set an attribute on the component reference. For example, if your component includes a reference to `<ui:button>`:

```
<ui:button label="Save">
    <aura:set attribute="buttonTitle" value="Click to save the record"/>
</ui:button>
```

This is equivalent to:

```
<ui:button label="Save" buttonTitle="Click to save the record" />
```

The latter syntax without `aura:set` makes more sense in this simple example. You can also use this simpler syntax in component references to set values for attributes that are inherited from parent components.

`aura:set` is more useful when you want to set markup as the attribute value. For example, this sample specifies the markup for the `else` attribute in the `aura:if` tag.

```
<aura:component>
    <aura:attribute name="display" type="Boolean" default="true"/>
    <aura:if isTrue="{!v.display}">
        Show this if condition is true
        <aura:set attribute="else">
            <ui:button label="Save" press="{!c.saveRecord}" />
        </aura:set>
    </aura:if>
</aura:component>
```

SEE ALSO:

[Setting Attributes Inherited from a Super Component](#)

Setting Attributes Inherited from an Interface

To set the value of an attribute inherited from an interface, redefine the attribute in the component and set its default value. Let's look at an example with the `docsample:myIntf` interface.

```
<!--docsample:myIntf-->
<aura:interface>
    <aura:attribute name="myBoolean" type="Boolean" default="true" />
</aura:interface>
```

This component implements the interface and sets `myBoolean` to `false`.

```
<!--docsample:myIntfImpl-->
<aura:component implements="docsample:myIntf">
    <aura:attribute name="myBoolean" type="Boolean" default="false" />

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

Component Reference

Reuse or extend out-of-the-box components for Salesforce1 or for your Lightning apps.

aura:component

The root of the component hierarchy. Provides a default rendering implementation.

Components are the functional units of Aura, which encapsulate modular and reusable sections of UI. They can contain other components or HTML markup. The public parts of a component are its attributes and events. Aura provides out-of-the-box components in the `aura` and `ui` namespaces.

Every component is part of a namespace. For example, the `button` component is saved as `button.cmp` in the `ui` namespace can be referenced in another component with the syntax `<ui:button label="Submit"/>`, where `label="Submit"` is an attribute setting.

To create a component, follow this syntax.

```
<aura:component>
  <!-- Optional coponent attributes here -->
  <!-- Optional HTML markup -->
  <div class="container">
    Hello world!
    <!-- Other components -->
  </div>
</aura:component>
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	

aura:expression

Renders the value to which an expression evaluates. Creates an instance of this component which renders the referenced "property reference value" set to the value attribute when expressions are found in free text or markup.

An expression is any set of literal values, variables, sub-expressions, or operators that can be resolved to a single value. It is used for dynamic output or passing a value into components by assigning them to attributes.

The syntax for an expression is `{!expression}`. `expression` is evaluated and dynamically replaced when the component is rendered or when the value is used by the component. The resulting value can be a primitive (integer, string, and so on), a boolean, a JavaScript or Aura object, an Aura component or collection, a controller method such as an action method, and other useful results.

An expression uses a value provider to access data and can also use operators and functions for more complex expressions. Value providers include `m` (data from model), `v` (attribute data from component), and `c` (controller action). This example show an expression `{!v.num}` whose value is resolved by the attribute `num`.

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

Attributes

Attribute Name	Attribute Type	Description	Required?
value	String	The expression to evaluate and render.	

`aura:html`

A meta component that represents all html elements. Any html found in your markup causes the creation of one of these.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
HTMLAttributes	HashMap	A Map of attributes to set on the html element.	
tag	String	The name of the html element that should be rendered.	

`aura:if`

Conditionally instantiates and renders either the body or the components in the else attribute.

aura:if evaluates the isTrue expression on the server and instantiates components in either its body or else attribute.

This example renders the body since isTrue evaluates to true.

```
<aura:attribute name="display" type="Boolean" default="true"/>
<aura:if isTrue="{!v.display}">
  Show this if true
  <aura:set attribute="else">
    Show this if false
  </aura:set>
</aura:if>
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	ComponentDefRef[]	The components to render when isTrue evaluates to true.	Yes
else	ComponentDefRef[]	The alternative to render when isTrue evaluates to false, and the body is not rendered. Should always be set using the aura:set tag.	
isTrue	Boolean	An expression that must be fulfilled in order to display the body.	Yes

aura:iteration

Renders a view of a collection of items. Supports iterations containing components that can be created exclusively on the client-side.

`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. It also supports iterations containing components that are created exclusively on the client-side or components that have server-side dependencies.

This example shows a basic way to use `aura:iteration` exclusively on the client-side.

```
<aura:component>
  <aura:iteration items="1,2,3,4,5" var="item">
        <meter value="{!item / 5}"/><br/>
    </aura:iteration>
</aura:component>
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	ComponentDefRef[]	Template to use when creating components for each iteration.	Yes
end	Integer	The index of the collection to stop at (exclusive)	
indexVar	String	The name of variable to use for the index of each item inside the iteration	
items	List	The collection of data to iterate over	Yes
loaded	Boolean	True if the iteration has finished loading the set of templates.	
start	Integer	The index of the collection to start at (inclusive)	
template	ComponentDefRef[]	The template that is used to generate components. By default, this is set from the body markup on first load.	
var	String	The name of the variable to use for each item inside the iteration	Yes

aura:renderIf

This component allows you to conditionally render its contents. It renders its body only if isTrue evaluates to true. The else attribute allows you to render an alternative when isTrue evaluates to false.

The expression in `isTrue` is re-evaluated every time any value used in the expression changes. When the results of the expression change, it triggers a re-rendering of the component. Use `aura:if` instead if you want to instantiate the components in either its body or the else attribute, but not both. Otherwise, use `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.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	

Attribute Name	Attribute Type	Description	Required?
else	Component[]	The alternative content to render when isTrue evaluates to false, and the body is not rendered. Set using the <aura:set> tag.	
isTrue	Boolean	An expression that must evaluate to true to display the body of the component.	Yes

aura:text

Renders plain text. When any free text (not a tag or attribute value) is found in markup, an instance of this component is created with the value attribute set to the text found in the markup.

Attributes

Attribute Name	Attribute Type	Description	Required?
value	String	The String to be rendered.	

aura:unescapedHtml

The value assigned to this component will be rendered as-is, without altering its contents. It's intended for outputting pre-formatted HTML, for example, where the formatting is arbitrary, or expensive to calculate. The body of this component is ignored, and won't be rendered. Warning: this component outputs value as unescaped HTML, which introduces the possibility of security vulnerabilities in your code. You must sanitize user input before rendering it unescaped, or you will create a cross-site scripting (XSS) vulnerability. Only use <aura:unescapedHtml> with trusted or sanitized sources of data.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of <aura:unescapedHtml> is ignored and won't be rendered.	
value	String	The string that should be rendered as unescaped HTML.	

force:inputField

An abstract component that provides a concrete type-specific input component implementation based on the data to which it is bound.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	

Attribute Name	Attribute Type	Description	Required?
class	String	The CSS style used to display the field.	
errorComponent	Component[]	A component which is responsible for displaying the error message.	
required	Boolean	Specifies whether this field is required or not.	
value	Object	Data value of Salesforce field to which to bind.	

force:outputField

An abstract component that provides a concrete type-specific output component implementation based on the data to which it is bound.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
value	Object	Data value of Salesforce field to which to bind.	

force:recordEdit

Generates an editable view of the specified Salesforce record.

A `force:recordEdit` component represents the record edit UI for the specified `recordId`. This example displays the record edit UI and a button, which when pressed saves the record.

```
<force:recordEdit aura:id="edit" recordId="a02D0000006V8Ni"/>
<ui:button label="Save" press="{!c.save}"/>
```

This client-side controller fires the `recordSave` event, which saves the record.

```
save : function(component, event, helper) {
component.find("edit").get("e.recordSave").fire();
// Update the component
helper.getRecords(component);
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	

Attribute Name	Attribute Type	Description	Required?
recordId	String	The Id of the record to load, optional if record attribute is specified.	

Events

Event Name	Event Type	Description
recordSave	COMPONENT	Record save request
recordSaveSuccess	COMPONENT	Indicates that the record has been successfully saved.

`force:recordView`

Generates a view of the specified Salesforce record.

A `force:recordView` component represents a read-only view of a record. You can display the record view using different layout types. By default, the record view uses the full layout to display all fields of the record. The mini layout displays the name field and any associated parent record field. This example shows a record view with a mini layout.

```
<force:recordView recordId="a02D0000006V8Ov" type="MINI"/>
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
record	SObjectRow	The record (SObject) to load, optional if recordId attribute is specified.	
recordId	String	The Id of the record to load, optional if record attribute is specified.	
type	String	The type of layout to use to display the record. Possible values: FULL, MINI. The default is FULL.	

`forceChatter:feed`

Represents a Chatter Feed

A `forceChatter:feed` component represents a feed that's specified by its type. Use the `type` attribute to display a specific feed type. For example, set `type="groups"` to display the feed from all groups the context user either owns or is a member of.

```
<aura:component implements="force:appHostable">
    <forceChatter:feed type="groups"/>
</aura:component>
```

You can also display a feed depending on the type selected. This example provides a drop-down menu that controls the type of feed to display.

```
<aura:component implements="force:appHostable">
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
   <aura:attribute name="type" type="String" default="News" description="The type of feed"
 access="GLOBAL"/>
    <aura:attribute name="types" type="String[]"
                    default="Bookmarks,Company,Files,Groups,Home,News,People"
                    description="A list of feed types"/>
   <h1>My Feeds</h1>
   <ui:inputSelect aura:id="typeSelect" change="{!c.onChangeType}" label="Type"/>
    <div aura:id="feedContainer" class="feed-container">
        <forceChatter:feed />
    </div>
</aura:component>
```

The `types` attribute specifies the feed types, which are set on the `ui:inputSelect` component during component initialization. When a user selects a feed type, the feed is dynamically created and displayed.

```
({
    // Handle component initialization
    doInit : function(component, event, helper) {
        var type = component.get("v.type");
        var types = component.get("v.types");
        var typeOpts = new Array();

        // Set the feed types on the ui:inputSelect component
        for (var i = 0; i < types.length; i++) {
           typeOpts.push({label: types[i], value: types[i], selected: types[i] === type});

        }
        component.find("typeSelect").set("v.options", typeOpts);
    },

 onChangeType : function(component, event, helper) {
        var typeSelect = component.find("typeSelect");
        var type = typeSelect.get("v.value");
        component.set("v.type", type);

        // Dynamically create the feed with the specified type
        $A.componentService.newComponentAsync(
            this,
            function(feed){
                var feedContainer = component.find("feedContainer");
                feedContainer.set("v.body", feed);
            },
            {
            componentDef : "markup://forceChatter:feed",
            attributes : {
                values : {
                    type: type
                }
            }
            }
```

196

```
        );
    }
})
```

The feed is supported for the Salesforce1 app only. You can include this feed in a component and access it in the Salesforce1 app. If used outside of Salesforce1, you may encounter issues with dead links, unhandled events, and missing styles. For a list of feed types, see Working with Feeds and Feed Elements in the Chatter REST API Developer's Guide.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
subjectId	String	For most feeds tied to an entity, this is used specified the desired entity. Defaults to the current user if not specified	
type	String	The strategy used to find items associated with the subject. Valid values include: News, Home, Record, To.	

ltng:require

Loads scripts and stylesheets while maintaining dependency order, once and only once injection, across multiple components.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
scripts	String[]	The set of style sheets in dependency order that will be loaded.	
styles	String[]	The set of scripts in dependency order that will be loaded.	

Events

Event Name	Event Type	Description
afterScriptsLoaded	COMPONENT	Fired when ltng:require has loaded all scripts listed in ltng:require.scripts

ui:actionMenuItem

A menu item that triggers an action. This component is nested in a ui:menu component.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
hideMenuAfterSelected	Boolean	Set to true to hide menu after the menu item is selected.	
label	String	The text displayed on the component.	
selected	Boolean	The status of the menu item. True means this menu item is selected; False is not selected.	
type	String	The concrete type of the menu item. Accepted values are 'action', 'checkbox', 'radio', 'separator' or any namespaced component descriptor, e.g. ns:xxxxmenuItem.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.

`ui:button`

Represents a button element.

A `ui:button` component represents a button element that executes an action defined by a controller. Clicking the button triggers the client-side controller method set for the `press` event. The button can be created in several ways.

A text-only button has only the required `label` attribute set on it.

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

An image-only button uses both the `label` and `labelClass` attributes with CSS.

```
<!-- 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. To create a button with both image and text, use the `label` attribute and add styles for the button.

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

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

The previous 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>
```

This example shows a button that displays the input value you enter.

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

```
({
    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);
    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
accesskey	String	The keyboard access key that puts the button in focus. When the button is in focus, pressing Enter clicks the button.	
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
buttonTitle	String	The text displayed in a tooltip when the mouse pointer hovers over the button.	
buttonType	String	Specifies the type attribute in the HTML input element. Default value is "button".	
class	String	A CSS style to be attached to the button. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether this button should be displayed in a disabled state. Disabled buttons can't be clicked. Default value is "false".	
label	String	The text displayed on the button. Corresponds to the value attribute of the rendered HTML input element.	Yes
labelClass	String	A CSS style to be attached to the label. This style is added in addition to base styles output by the component.	

Events

Event Name	Event Type	Description
press	COMPONENT	Indicates that the component has been pressed.

`ui:checkboxMenuItem`

A menu item with a checkbox that supports multiple selection and can be used to invoke an action. This component is nested in a ui:menu component.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	

Attribute Name	Attribute Type	Description	Required?
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
~~hideMenuAfterSelected~~	Boolean	Set to true to hide menu after the menu item is selected.	
label	String	The text displayed on the component.	
selected	Boolean	The status of the menu item. True means this menu item is selected; False is not selected.	
type	String	The concrete type of the menu item. Accepted values are 'action', 'checkbox', 'radio', 'separator' or any namespaced component descriptor, e.g. ns:xxxxmenuItem.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.

ui:inputCheckbox

Represents a checkbox. Its behavior can be configured using events such as click and change.

A `ui:inputCheckbox` component represents a checkbox whose state is controlled by the `value` and `disabled` attributes. It's rendered as an HTML `input` tag of type `checkbox`. To render the output from a `ui:inputCheckbox` component, use the `ui:outputCheckbox` component.

This is a basic set up of a checkbox.

```
<ui:inputCheckbox label="Reimbursed?"/>
```

This example results in the following HTML.

```
<div class="uiInput uiInputCheckbox">
  <label class="uiLabel-left uiLabel">
    <span>Reimbursed?</span>
  </label>
  <input type="checkbox" class="uiInput uiInputCheckbox">
</div>
```

The `value` attribute controls the state of a checkbox, and events such as `click` and `change` determine its behavior. This example updates the checkbox CSS class on a click event.

```
<!-- Component Markup -->
<ui:inputCheckbox label="Color me" click="{!c.update}"/>

/** Client-Side Controller **/
update : function (cmp, event) {
  var elem = event.getSource().getElement();
  $A.util.toggleClass(elem, "red");
}
```

This example retrieves the value of a `ui:inputCheckbox` component.

```
<aura:component>
 <aura:attribute name="myBool" type="Boolean" default="true"/>
 <ui:inputCheckbox aura:id="checkbox" label="Select?" change="{!c.onCheck}"/>
 <p>Selected:</p>
 <p><ui:outputText class="result" aura:id="checkResult" value="false" /></p>
 <p>The following checkbox uses a component attribute to bind its value.</p>
 <ui:outputCheckbox aura:id="output" value="{!v.myBool}"/>
</aura:component>
```

```
({
  onCheck: function(cmp, evt) {
   var checkCmp = cmp.find("checkbox");
   resultCmp = cmp.find("checkResult");
   resultCmp.set("v.value", ""+checkCmp.get("v.value"));

  }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
label	String	The text displayed on the component.	
labelClass	String	The CSS class of the label component	
name	String	The name of the component.	
required	Boolean	Specifies whether the input is required. Default value is "false".	
requiredIndicatorClass	String	The CSS class of the required indicator component	
text	String	The input value attribute.	
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change,click".	
value	Boolean	Indicates whether the status of the option is selected. Default value is "false".	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.

Event Name	Event Type	Description
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

ui:inputCurrency

An input field for entering a currency.

A `ui:inputCurrency` component represents an input field for a number as a currency, which is rendered as an HTML `input` tag of type `text`. The browser's locale is used by default. To render the output from a `ui:inputCurrency` component, use the `ui:outputCurrency` component.

This is a basic set up of a `ui:inputCurrency` component, which renders an input field with the value `$50.00` when the browser's currency locale is `$`.

```
<ui:inputCurrency label="Amount" class="field" value="50"/>
```

This example results in the following HTML.

```
<div class="uiInput">
    <label class="uiLabel-left uiLabel">
        <span>Amount</span>
    </label>
    <input class="field" max="99999999999999" step="1" type="text" min="-99999999999999">
</div>
```

To override the browser's locale, set the new format on the `v.format` attribute of the `ui:inputCurrency` component. This example renders an input field with the value `£50.00`.

```
var curr = component.find("amount");
curr.set("v.format", '£#,###.00');
```

This example retrieves the value of a `ui:inputCurrency` component and displays it using `ui:outputCurrency`.

```
<aura:component>
    <ui:inputCurrency aura:id="amount" label="Amount" class="field" value="50"/>
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
  You entered: <ui:outputCurrency aura:id="oCurrency" value=""/>
```

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

```
({
  setOutput : function(component, event, helper) {
  var el = component.find("msg");
      $A.util.removeClass(el.getElement(), 'hide');

        var amount = component.find("amount").get("v.value");
        var oCurrency = component.find("oCurrency");
        oCurrency.set("v.value", amount);
  }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
format	String	The format of the number. For example, format=".00" displays the number followed by two decimal places. If not specified, the Locale default format will be used.	
label	String	The text of the label component	
labelClass	String	The CSS class of the label component	
maxlength	Integer	The maximum number of characters that can be typed into the input field. Corresponds to the maxlength attribute of the rendered HTML input element.	
placeholder	String	Text that is displayed when the field is empty, to prompt the user for a valid entry.	
required	Boolean	Specifies whether the input is required. Default value is "false".	
requiredIndicatorClass	String	The CSS class of the required indicator component	
size	Integer	The width of the input field, in characters. Corresponds to the size attribute of the rendered HTML input element.	
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change".	
value	BigDecimal	The input value of the number.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

ui:inputDate

An input field for entering a date.

A `ui:inputDate` component represents a date input field, which is rendered as an HTML `input` tag of type `text`. The value is displayed in the default format specified language locale of the browser.

This is a basic set up of a date field with a date picker, which displays the field value `1/30/2014`. Specifying `format="MMMM d, yyyy"` renders the field value as `January 30, 2014`.

```
<ui:inputDate aura:id="dateField" label="Birthday" value="2014-01-30"
displayDatePicker="true"/>
```

This example results in the following HTML.

```
<div class="uiInput uiInputDate">
  <label class="uiLabel-left uiLabel">
```

```
    <span>Birthday</span>
  </label>
  <input placeholder="M/d/yyyy" type="text" class="uiInput uiInputDate">
  <a class="datePicker-openIcon" aria-haspopup="true">
    <span class="assistiveText">Date Picker</span>
  </a>
  <a class="clearIcon">
    <span class="assistiveText">Clear Button</span>
  </a>
  <div class="uiDatePicker">
    <!--Date picker set to visible when icon is clicked-->
  </div>
</div>
```

This example sets today's date on a `ui:inputDate` component, retrieves its value, and displays it using `ui:outputDate`. The `init` handler initializes and sets the date on the component.

```
<aura:component>
 <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
 <aura:attribute name="today" type="Date" default=""/>

    <ui:inputDate aura:id="expdate" label="Today's Date" class="field" value="{!v.today}"
 displayDatePicker="true" />
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
  You entered: <ui:outputDate aura:id="oDate" value="" />
 </div>
</aura:component>
```

```
({
    doInit : function(component, event, helper) {
        var today = new Date();
        component.set('v.today', today.getFullYear() + "-" + (today.getMonth() + 1) + "-"
 + today.getDate());
        component.set('v.deadline', today);
    },

    setOutput : function(component, event, helper) {
     var el = component.find("msg");
     $A.util.removeClass(el.getElement(), 'hide');
        var expdate = component.find("expdate").get("v.value");

        var oDate = component.find("oDate");
        oDate.set("v.value", expdate);

    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
displayDatePicker	Boolean	Indicate if ui:datePicker is displayed.	
format	String	The java.text.SimpleDateFormat style format string.	
label	String	The text of the label component	
labelClass	String	The CSS class of the label component	
langLocale	String	The language locale used to format date time.	
required	Boolean	Specifies whether the input is required. Default value is "false".	
requiredIndicatorClass	String	The CSS class of the required indicator component	
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change".	
value	Date	The input value of the date/time.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.

Event Name	Event Type	Description
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

`ui:inputDateTime`

An input field for entering a date and time.

A `ui:inputDateTime` component represents a date and time input field, which is rendered as an HTML `input` tag of type `text`. The value is displayed in the default format specified by the language locale of the browser.

This is a basic set up of a date and time field with a date picker, which displays the current date and time in the format `7/29/2014 1:11 PM`.

```
<!-- Component markup -->
<aura:attribute name="today" type="DateTime" />
<ui:inputDateTime aura:id="expdate" label="Expense Date" class="form-control"
   value="{!v.today}" displayDatePicker="true" />

/** Client-Side Controller **/
 var today = new Date();
component.set("v.today", today);
```

This example results in the following HTML.

```
<div class="uiInput uiInputDateTime">
<label class="uiLabel-left uiLabel">
    <span>Expense Date</span>
</label>
<input class="form-control uiInput uiInputDateTime" placeholder="M/d/yyyy h:mm a"
type="text">
<a class="datePicker-openIcon" aria-haspopup="true">
    <span class="assistiveText">Date Picker</span>
</a>
<a class="clearIcon" href="javascript:void(0);">
    <span class="assistiveText">Clear Button</span>
</a>
<div class="uiDatePicker">
    <!-- Date picker set to visible when icon is clicked -->
</div>
```

209

This example retrieves the value of a `ui:inputDateTime` component and displays it using `ui:outputDateTime`.

```
<aura:component>
 <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
 <aura:attribute name="today" type="Date" default=""/>

    <ui:inputDateTime aura:id="today" label="Time" class="field" value=""
displayDatePicker="true" />
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

    <div aura:id="msg" class="hide">
  You entered: <ui:outputDateTime aura:id="oDateTime" value=""  />
 </div>
</aura:component>
```

```
({
    doInit : function(component, event, helper) {
        var today = new Date();
        component.set('v.today', today.getFullYear() + "-" + (today.getMonth() + 1) + "-"
 + today.getDate());
    },

    setOutput : function(component, event, helper) {
     var el = component.find("msg");
     $A.util.removeClass(el.getElement(), 'hide');

        var todayVal = component.find("today").get("v.value");
        var oDateTime = component.find("oDateTime");
        oDateTime.set("v.value", todayVal);

    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
displayDatePicker	Boolean	Indicate if ui:datePicker is displayed.	
format	String	The java.text.SimpleDateFormat style format string.	
label	String	The text of the label component	
labelClass	String	The CSS class of the label component	

Attribute Name	Attribute Type	Description	Required?
langLocale	String	The language locale used to format date time.	
required	Boolean	Specifies whether the input is required. Default value is "false".	
requiredIndicatorClass	String	The CSS class of the required indicator component	
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change".	
value	String	The input value of the date/time.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

`ui:inputDefaultError`

The default implementation of field-level errors, which iterates over the value and displays the message.

`ui:inputDefaultError` is the default error handling for your input components. This component displays as a list of errors below the field. Field-level error messages can be added using `addErrors()`. You can use the default error handling by setting the input value to false and adding the error message. For example, this component validates if the input is a number.

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

This client-side controller displays an error if the input is not a number.

```
doAction : function(component, event) {
    var inputCmp = cmp.find("inputCmp");
    var value = inputCmp.get("v.value");
    if (isNaN(value)) {
        inputCmp.setValue("v.value", false);
        inputCmp.addErrors("v.value", [{message:"Input not a number: " + value}]);
    } else {
        //clear error
        inputCmp.setValid("v.value", true);
    }
}
```

Alternatively, you can provide your own `ui:inputDefaultError` component. This example returns an error message if the `warnings` attribute contains any messages.

```
<aura:component>
      <aura:attribute name="warnings" type="String[]" description="Warnings for input
text"/>
    Enter a number: <ui:inputNumber aura:id="inputCmp" label="number"/>
    <ui:button label="Submit" press="{!c.doAction}"/>
    <ui:inputDefaultError aura:id="number" value="{!v.warnings}" />
</aura:component>
```

This client-side controller diplays an error by adding a string to the `warnings` attribute.

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

    // is input numeric?
    if (isNaN(value)) {
       component.set("v.warnings", "Input is not a number");
    } else {
       // clear error
       component.set("v.warnings", null);
    }
}
```

This example shows a `ui:inputText` component with the default error handling, and a corresponding `ui:outputText` component for text rendering.

```
<aura:component>
 <ui:inputText aura:id="color" label="Enter some text: " placeholder="Blue" />
 <ui:button label="Validate" press="{!c.checkInput}" />
 <ui:outputText aura:id="outColor" value="" class="text"/>
</aura:component>
```

```
({
    checkInput : function(cmp, evt) {
     var colorCmp = cmp.find("color");
        var myColor = colorCmp.get("v.value");
        var myOutput = cmp.find("outColor");
        var greet = "You entered: " + myColor;
        myOutput.set("v.value", greet);

        if (!myColor) {
            colorCmp.setValid("v.value", false);
            colorCmp.addErrors("v.value", [{message:"Enter some text"}]);
        }
        else {
            //clear error
            if(!colorCmp.isValid("v.value")){
                colorCmp.setValid("v.value", true);
            }
        }
    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
value	String[]	The list of error messages to be displayed.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.

Event Name	Event Type	Description
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

`ui:inputEmail`

Represents an input field for entering an email address.

A `ui:inputEmail` component represents an email input field, which is rendered as an HTML `input` tag of type `email`. To render the output from a `ui:inputEmail` component, use the `ui:outputEmail` component.

This is a basic set up of an email field.

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

This example results in the following HTML.

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

This example retrieves the value of a `ui:inputEmail` component and displays it using `ui:outputEmail`.

```
<aura:component>
    <ui:inputEmail aura:id="email" label="Email" class="field" value="manager@email.com"/>

    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
     You entered: <ui:outputEmail aura:id="oEmail" value="Email" />
 </div>

</aura:component>
```

```
({
    setOutput : function(component, event, helper) {
     var el = component.find("msg");
     $A.util.removeClass(el.getElement(), 'hide');

        var email = component.find("email").get("v.value");
        var oEmail = component.find("oEmail");
        oEmail.set("v.value", email);
```

```
    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
label	String	The text of the label component	
labelClass	String	The CSS class of the label component	
maxlength	Integer	The maximum number of characters that can be typed into the input field. Corresponds to the maxlength attribute of the rendered HTML input element.	
placeholder	String	Text that is displayed when the field is empty, to prompt the user for a valid entry.	
required	Boolean	Specifies whether the input is required. Default value is "false".	
requiredIndicatorClass	String	The CSS class of the required indicator component	
size	Integer	The width of the input field, in characters. Corresponds to the size attribute of the rendered HTML input element.	
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change".	
value	String	The value currently in the input field.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.

Event Name	Event Type	Description
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

ui:inputNumber

An input field for entering a number, taking advantage of client input assistance and validation when available.

A ui:inputNumber component represents a number input field, which is rendered as an HTML input tag of type text. 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 uiInputNumber">
<label class="uiLabel-left uiLabel">
    <span>Age</span>
</label>
<input max="99999999999999" step="1" type="text"
      min="-99999999999999" class="uiInput uiInputNumber">
</div>
```

To render the output from a ui:inputNumber component, use the ui:inputNumber component. When providing a number value with commas, use type="integer". This example returns 100,000.

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

For `type="string"`, provide the number without commas for the output to be formatted accordingly. This example also returns `100,000`.

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

Specifying `format="#,##0,000.00#"` returns a formatted number value like `10,000.00`.

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

This example retrieves the value of a `ui:inputNumber` component, validates the input, and displays it using `ui:outputNumber`.

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

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

        var myOutput = component.find("outNum");

        myOutput.set("v.value", value);

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

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	

217

Attribute Name	Attribute Type	Description	Required?
format	String	The format of the number. For example, format=".00" displays the number followed by two decimal places. If not specified, the Locale default format will be used.	
label	String	The text of the label component	
labelClass	String	The CSS class of the label component	
maxlength	Integer	The maximum number of characters that can be typed into the input field. Corresponds to the maxlength attribute of the rendered HTML input element.	
placeholder	String	Text that is displayed when the field is empty, to prompt the user for a valid entry.	
required	Boolean	Specifies whether the input is required. Default value is "false".	
requiredIndicatorClass	String	The CSS class of the required indicator component	
size	Integer	The width of the input field, in characters. Corresponds to the size attribute of the rendered HTML input element.	
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change".	
value	BigDecimal	The input value of the number.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.

Event Name	Event Type	Description
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

`ui:inputPhone`

Represents an input field for entering a telephone number.

A `ui:inputPhone` component represents an input field for entering a phone number, which is rendered as an HTML `input` tag of type `tel`. To render the output from a `ui:inputPhone` component, use the `ui:outputPhone` component.

This example shows a phone field, which displays the specified phone number.

```
<ui:inputPhone label="Phone" value="415-123-4567" />
```

The previous example results in the following HTML.

```
<div class="uiInput uiInputPhone">
    <label class="uiLabel-left uiLabel">
        <span>Phone</span>
    </label>
    <input class="uiInput uiInputPhone" type="tel">
</div>
```

This example retrieves the value of a `ui:inputPhone` component and displays it using `ui:outputPhone`.

```
<aura:component>
    <ui:inputPhone aura:id="phone" label="Phone Number" class="field" value="415-123-4567"
 />
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
  You entered: <ui:outputPhone aura:id="oPhone" value="" />
 </div>
</aura:component>
```

```
({

    setOutput : function(component, event, helper) {
     var el = component.find("msg");
     $A.util.removeClass(el.getElement(), 'hide');

         var phone = component.find("phone").get("v.value");
```

```
        var oPhone = component.find("oPhone");
        oPhone.set("v.value", phone);
    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
label	String	The text of the label component	
labelClass	String	The CSS class of the label component	
maxlength	Integer	The maximum number of characters that can be typed into the input field. Corresponds to the maxlength attribute of the rendered HTML input element.	
placeholder	String	Text that is displayed when the field is empty, to prompt the user for a valid entry.	
required	Boolean	Specifies whether the input is required. Default value is "false".	
requiredIndicatorClass	String	The CSS class of the required indicator component	
size	Integer	The width of the input field, in characters. Corresponds to the size attribute of the rendered HTML input element.	
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change".	
value	String	The value currently in the input field.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.

220

Event Name	Event Type	Description
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

ui:inputRadio

The radio button used in the input.

A `ui:inputRadio` component represents a radio button whose state is controlled by the `value` and `disabled` attributes. It's rendered as an HTML `input` tag of type `radio`. To group your radio buttons together, specify the `name` attribute with a unique name.

This is a basic set up of a radio button.

```
<ui:inputRadio label="Yes"/>
```

This example results in the following HTML.

```
<div class="uiInput uiInputRadio">
    <label class="uiLabel-left uiLabel">
        <span>Yes</span>
    </label>
    <input type="radio">
</div>
```

This example retrieves the value of a selected `ui:inputRadio` component.

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

221

```
      </aura:iteration>

   <b>Selected Item:</b>
   <p><ui:outputText class="result" aura:id="radioResult" value="" /></p>

   <b>Radio Buttons - Group</b>
   <ui:inputRadio aura:id="r0" name="others" label="Prospecting" change="{!c.onGroup}"/>
   <ui:inputRadio aura:id="r1" name="others" label="Qualification" change="{!c.onGroup}"
value="true"/>
  <ui:inputRadio aura:id="r2" name="others" label="Needs Analysis" change="{!c.onGroup}"/>

   <ui:inputRadio aura:id="r3" name="others" label="Closed Lost" change="{!c.onGroup}"/>
   <b>Selected Items:</b>
   <p><ui:outputText class="result" aura:id="radioGroupResult" value="" /></p>

</aura:component>
```

```
({
 onRadio: function(cmp, evt) {
   var elem = evt.getSource().getElement();
   var selected = elem.textContent;
   resultCmp = cmp.find("radioResult");
   resultCmp.set("v.value", selected);
  },

  onGroup: function(cmp, evt) {
   var elem = evt.getSource().getElement();
   var selected = elem.textContent;
   resultCmp = cmp.find("radioGroupResult");
   resultCmp.set("v.value", selected);
  }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether this radio button should be displayed in a disabled state. Disabled radio buttons can't be clicked. Default value is "false".	
label	String	The text displayed on the component.	
labelClass	String	The CSS class of the label component	
name	String	The name of the component.	
required	Boolean	Specifies whether the input is required. Default value is "false".	

Attribute Name	Attribute Type	Description	Required?
requiredIndicatorClass	String	The CSS class of the required indicator component	
text	String	The input value attribute.	
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change".	
value	Boolean	Indicates whether the status of the option is selected. Default value is "false".	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

`ui:inputRichText`

An input field for entering rich text.

By default, `ui:inputRichText` renders a WYSIWYG editor for entering rich text. Setting `isRichText="false"` uses the `ui:inputTextArea` component instead of a WYSIWYG editor.

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

`ui:outputRichText` supports a list of HTML tags that are supported by CKEditor. Tags such as `<script>` are removed.

This example shows a text area and WYSIWYG editor.

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

```
({
 getInput : function(cmp) {

  var userInput = cmp.find("inputRT").get("v.value");
  var output = cmp.find("outputRT");
  output.set("v.value", userInput);
 }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
cols	Integer	The width of the text area, which is defined by the number of characters to display in a single row at a time. Default value is "20".	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
height	String	The height of the editing area (that includes the editor content). This can be an integer, for pixel sizes, or any CSS-defined length unit.	
label	String	The text of the label component	

Attribute Name	Attribute Type	Description	Required?
labelClass	String	The CSS class of the label component	
maxlength	Integer	The maximum number of characters that can be typed into the input field. Corresponds to the maxlength attribute of the rendered HTML textarea element.	
placeholder	String	The text that is displayed by default.	
readonly	Boolean	Specifies whether the text area should be rendered as read-only. Default value is "false".	
required	Boolean	Specifies whether the input is required. Default value is "false".	
requiredIndicatorClass	String	The CSS class of the required indicator component	
resizable	Boolean	Specifies whether or not the textarea should be resizable. Defaults to true.	
rows	Integer	The height of the text area, which is defined by the number of rows to display at a time. Default value is "2".	
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change".	
value	String	The value currently in the input field.	
width	String	The editor UI outer width. This can be an integer, for pixel sizes, or any CSS-defined unit. If isRichText is set to false, use the cols attribute instead.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.

Event Name	Event Type	Description
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

ui:inputSecret

An input field for entering secret text with type password.

A `ui:inputSecret` component represents a password field, which is rendered as an HTML `input` tag of type `password`.

This is a basic set up of a password field.

```
<ui:inputSecret aura:id="secret" label="Pin" class="field" value="123456"/>
```

This example results in the following HTML.

```
<div class="uiInput uiInputSecret">
    <label class="uiLabel-left uiLabel">
        <span>Pin</span>
    </label>
    <input class="field" type="password">
</div>
```

This example displays a `ui:inputSecret` component with a default value.

```
<aura:component>
    <ui:inputSecret aura:id="secret" label="Pin" class="field" value="123456"/>
</aura:component>
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	

226

Attribute Name	Attribute Type	Description	Required?
label	String	The text of the label component	
labelClass	String	The CSS class of the label component	
maxlength	Integer	The maximum number of characters that can be typed into the input field. Corresponds to the maxlength attribute of the rendered HTML input element.	
placeholder	String	Text that is displayed when the field is empty, to prompt the user for a valid entry.	
required	Boolean	Specifies whether the input is required. Default value is "false".	
requiredIndicatorClass	String	The CSS class of the required indicator component	
size	Integer	The width of the input field, in characters. Corresponds to the size attribute of the rendered HTML input element.	
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change".	
value	String	The value currently in the input field.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.

Event Name	Event Type	Description
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

ui:inputSelect

Represents a drop-down list with options.

A `ui:inputSelect` component is rendered as an HTML `select` element. It contains options, represented by the `ui:inputSelectOption` components. To enable multiple selections, set `multiple="true"`. To wire up any client-side logic when an input value is selected, use the `change` event.

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

Generating Options with `aura:iteration`

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

```
<aura:attribute name="contacts" type="String[]" default="Primary Contact, Secondary Contact,
 Other"/>
<ui:inputSelect>
    <aura:iteration items="{!v.contacts}" var="contact">
        <ui:inputSelectOption text="{!contact}" label="{!contact}"/>
    </aura:iteration>
</ui:inputSelect>
```

Generating Options Dynamically

Generate the options dynamically on component initialization using a controller-side action.

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

The following client-side controller generates options using the options attribute on the `ui:inputSelect` component. `v.options` takes in the list of objects and converts them into list options. The `opts` object constructs `InputOption` objects to create the `ui:inputSelectOptions` components within `ui:inputSelect`. Although the sample code generates the options during initialization, the list of options can be modified anytime when you manipulate the list in `v.options`. The component automatically updates itself and rerenders with the new options.

```
({
    doInit : function(cmp) {
        var opts = [
```

```
            { class: "optionClass", label: "Option1", value: "opt1", selected: "true" },
            { class: "optionClass", label: "Option2", value: "opt2" },
            { class: "optionClass", label: "Option3", value: "opt3" }

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

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

This example displays a drop-down list with single and multiple selection enabled, and another with dynamically generated list options. It retrieves the selected value of a `ui:inputSelect` component.

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

<div class="row">
<p class="title">Single Selection</p>
<ui:inputSelect class="single" aura:id="InputSelectSingle"
change="{!c.onSingleSelectChange}">

            <ui:inputSelectOption text="Any"/>
            <ui:inputSelectOption text="Open" value="true"/>
            <ui:inputSelectOption text="Closed"/>
            <ui:inputSelectOption text="Closed Won"/>
        <ui:inputSelectOption text="Prospecting"/>
            <ui:inputSelectOption text="Qualification"/>
            <ui:inputSelectOption text="Needs Analysis"/>
            <ui:inputSelectOption text="Closed Lost"/>
    </ui:inputSelect>
    <p>Selected Item:</p>
      <p><ui:outputText class="result" aura:id="singleResult" value="" /></p>
</div>

<div class="row">
    <p class="title">Multiple Selection</p>
    <ui:inputSelect multiple="true" class="multiple" aura:id="InputSelectMultiple"
change="{!c.onMultiSelectChange}">

            <ui:inputSelectOption text="Any"/>
            <ui:inputSelectOption text="Open"/>
            <ui:inputSelectOption text="Closed"/>
            <ui:inputSelectOption text="Closed Won"/>
            <ui:inputSelectOption text="Prospecting"/>
            <ui:inputSelectOption text="Qualification"/>
            <ui:inputSelectOption text="Needs Analysis"/>
            <ui:inputSelectOption text="Closed Lost"/>

    </ui:inputSelect>
    <p>Selected Items:</p>
     <p><ui:outputText class="result" aura:id="multiResult" value="" /></p>
```

```
</div>

<div class="row">
   <p class="title">Dynamic Option Generation</p>
   <ui:inputSelect label="Select me: " class="dynamic" aura:id="InputSelectDynamic"
change="{!c.onChange}" />
   <p>Selected Items:</p>
   <p><ui:outputText class="result" aura:id="dynamicResult" value="" /></p>
</div>

</aura:component>
```

```
({
    doInit : function(cmp) {
     // Initialize input select options
        var opts = [
            { "class": "optionClass", label: "Option1", value: "opt1", selected: "true"
},
            { "class": "optionClass", label: "Option2", value: "opt2" },
            { "class": "optionClass", label: "Option3", value: "opt3" }

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

    },

 onSingleSelectChange: function(cmp, evt) {
        var selectCmp = cmp.find("InputSelectSingle");
        resultCmp = cmp.find("singleResult");
        resultCmp.set("v.value", selectCmp.get("v.value"));
  },

  onMultiSelectChange: function(cmp, evt) {
        var selectCmp = cmp.find("InputSelectMultiple");
        resultCmp = cmp.find("multiResult");
        resultCmp.set("v.value", selectCmp.get("v.value"));
  },

  onChange: function(cmp, evt) {
   var dynamicCmp = cmp.find("InputSelectDynamic");
   resultCmp = cmp.find("dynamicResult");
   resultCmp.set("v.value", dynamicCmp.get("v.value"));
  }

})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	

Attribute Name	Attribute Type	Description	Required?
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
label	String	The text of the label component	
labelClass	String	The CSS class of the label component	
multiple	Boolean	Specifies whether the input is a multiple select. Default value is "false".	
options	List	A list of aura.components.ui.InputOption.	
required	Boolean	Specifies whether the input is required. Default value is "false".	
requiredIndicatorClass	String	The CSS class of the required indicator component	
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change".	
value	String	The value currently in the input field.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.

Event Name	Event Type	Description
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

ui:inputSelectOption

An HTML option element that is nested in a <ui:inputSelect> element. Denotes the available options in the list.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
label	String	The text displayed on the component.	
name	String	The name of the component.	
text	String	The input value attribute.	
value	Boolean	Indicates whether the status of the option is selected. Default value is "false".	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

Event Name	Event Type	Description
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.

ui:inputText

Represents an input field suitable for entering a single line of free-form text.

A `ui:inputText` component represents a text input field, which is rendered as an HTML `input` tag of type `text`. To render the output from a `ui:inputText` component, use the `ui:outputText` component.

This is a basic set up of a text field.

```
<ui:inputText label="Expense Name" value="My Expense" required="true"/>
```

This example results in the following HTML.

```
<div class="uiInput uiInputText">
  <label class="uiLabel-left uiLabel">
    <span>Expense Name</span>
    <span class="required">*</span>
  </label>
  <input required="required" class="uiInput uiInputText" type="text">
</div>
```

This example retrieves the value of a `ui:inputText` component and displays it using `ui:outputText`.

```
<aura:component>
 <ui:inputText aura:id="name" label="Enter some text" class="field" value="My Text"/>
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
  You entered: <ui:outputText aura:id="oName" value=""/>
 </div>
</aura:component>
```

```
({
  setOutput : function(component, event, helper) {
  var el = component.find("msg");
    $A.util.removeClass(el.getElement(), 'hide');

      var name = component.find("name").get("v.value");
      var oName = component.find("oName");
      oName.set("v.value", name);
```

233

```
    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
label	String	The text of the label component	
labelClass	String	The CSS class of the label component	
maxlength	Integer	The maximum number of characters that can be typed into the input field. Corresponds to the maxlength attribute of the rendered HTML input element.	
placeholder	String	Text that is displayed when the field is empty, to prompt the user for a valid entry.	
required	Boolean	Specifies whether the input is required. Default value is "false".	
requiredIndicatorClass	String	The CSS class of the required indicator component	
size	Integer	The width of the input field, in characters. Corresponds to the size attribute of the rendered HTML input element.	
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change".	
value	String	The value currently in the input field.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.

234

Event Name	Event Type	Description
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

`ui:inputTextArea`

An HTML textarea element that can be editable or read-only. Scroll bars may not appear on Chrome browsers in Android devices, but you can select focus in the textarea to activate scrolling.

A `ui:inputTextArea` component represents a multi-line text input control, which is rendered as an HTML `textarea` tag. To render the output from a `ui:inputTextArea` component, use the `ui:outputTextArea` component.

This is a basic set up of a `ui:inputTextArea` component.

```
<ui:inputTextArea aura:id="comments" label="Comments" value="My comments" rows="5"/>
```

This example results in the following HTML.

```
<div class="uiInput uiInputTextArea">
    <label class="uiLabel-left uiLabel">
        <span>Comments</span>
    </label>
    <textarea cols="20" rows="5">
    </textarea>
</div>
```

This example retrieves the value of a `ui:inputTextArea` component and displays it using `ui:outputTextArea`.

```
<aura:component>
    <ui:inputTextArea aura:id="comments" label="Comments"  value="My comments" rows="5"/>

    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>
```

235

```
    <div aura:id="msg" class="hide">
 You entered: <ui:outputTextArea aura:id="oTextarea" value=""/>
 </div>
</aura:component>
```

```
({
    setOutput : function(component, event, helper) {
     var el = component.find("msg");
     $A.util.removeClass(el.getElement(), 'hide');

        var comments = component.find("comments").get("v.value");
        var oTextarea = component.find("oTextarea");
        oTextarea.set("v.value", comments);
    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
cols	Integer	The width of the text area, which is defined by the number of characters to display in a single row at a time. Default value is "20".	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
label	String	The text of the label component	
labelClass	String	The CSS class of the label component	
maxlength	Integer	The maximum number of characters that can be typed into the input field. Corresponds to the maxlength attribute of the rendered HTML textarea element.	
placeholder	String	The text that is displayed by default.	
readonly	Boolean	Specifies whether the text area should be rendered as read-only. Default value is "false".	
required	Boolean	Specifies whether the input is required. Default value is "false".	
requiredIndicatorClass	String	The CSS class of the required indicator component	
resizable	Boolean	Specifies whether or not the textarea should be resizable. Defaults to true.	
rows	Integer	The height of the text area, which is defined by the number of rows to display at a time. Default value is "2".	

236

Attribute Name	Attribute Type	Description	Required?
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change".	
value	String	The value currently in the input field.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

ui:inputURL

An input field for entering a URL.

A ui:inputURL component represents an input field for a URL, which is rendered as an HTML input tag of type url. To render the output from a ui:inputURL component, use the ui:outputURL component.

This is a basic set up of a `ui:inputURL` component.

```
<ui:inputURL aura:id="url" label="Venue URL" class="field" value="http://www.myURL.com"/>
```

This example results in the following HTML.

```
<div class="uiInput uiInputText uiInputURL">
    <label class="uiLabel-left uiLabel">
        <span>Venue URL</span>
    </label>
    <input class="field" type="url">
</div>
```

This example retrieves the value of a `ui:inputURL` component and displays it using `ui:outputURL`.

```
<aura:component>
    <ui:inputURL aura:id="url" label="Venue URL" class="field" value="http://www.myURL.com"/>

    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>
 <div aura:id="msg" class="hide">
  You entered: <ui:outputURL aura:id="oURL" value=""/>
 </div>
</aura:component>
```

```
({
    setOutput : function(component, event, helper) {
     var el = component.find("msg");
     $A.util.removeClass(el.getElement(), 'hide');

        var url = component.find("url").get("v.value");
        var oURL = component.find("oURL");
        oURL.set("v.value", url);
        oURL.set("v.label", url);
    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
label	String	The text of the label component	
labelClass	String	The CSS class of the label component	

Attribute Name	Attribute Type	Description	Required?
maxlength	Integer	The maximum number of characters that can be typed into the input field. Corresponds to the maxlength attribute of the rendered HTML input element.	
placeholder	String	Text that is displayed when the field is empty, to prompt the user for a valid entry.	
required	Boolean	Specifies whether the input is required. Default value is "false".	
requiredIndicatorClass	String	The CSS class of the required indicator component	
size	Integer	The width of the input field, in characters. Corresponds to the size attribute of the rendered HTML input element.	
updateOn	String	Updates the component's value binding if the updateOn attribute is set to the handled event. Default value is "change".	
value	String	The value currently in the input field.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
validationError	COMPONENT	Indicates that the component has validation error(s).
paste	COMPONENT	Indicates that the user has pasted content from the clipboard.
change	COMPONENT	Indicates that the content of a component or the state has changed.
clearErrors	COMPONENT	Indicates that any validation error should be cleared.

Event Name	Event Type	Description
cut	COMPONENT	Indicates that the user has cut content to the clipboard.
copy	COMPONENT	Indicates that the user has copied content to the clipboard.

ui:menu

A dropdown menu list with a trigger that controls its visibility. Need to provide a menuTriggerLink and menuList component.

A `ui:menu` component contains a trigger and list items. You can wire up list items to actions in a client-side controller so that selection of the item triggers an action. This example shows a menu with list items, which when pressed updates the label on the trigger.

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Opportunity Status"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu">
            <ui:actionMenuItem aura:id="item1" label="Any"
click="{!c.updateTriggerLabel}"/>
            <ui:actionMenuItem aura:id="item2" label="Open" click="{!c.updateTriggerLabel}"
 disabled="true"/>
            <ui:actionMenuItem aura:id="item3" label="Closed"
click="{!c.updateTriggerLabel}"/>
            <ui:actionMenuItem aura:id="item4" label="Closed Won"
click="{!c.updateTriggerLabel}"/>
        </ui:menuList>
</ui:menu>
```

This client-side controller updates the trigger label when a menu item is clicked.

```
({
    updateTriggerLabel: function(cmp, event) {
        var triggerCmp = cmp.find("trigger");
        if (triggerCmp) {
            var source = event.getSource();
            var label = source.get("v.label");
            triggerCmp.set("v.label", label);
        }
    }
})
```

The dropdown menu and its menu items are hidden by default. You can change this by setting the `visible` attribute on the `ui:menuList` component to `true`. The menu items are shown only when you click the `ui:menuTriggerLink` component.

To use a trigger, which opens the menu, nest the `ui:menuTriggerLink` component in `ui:menu`. For list items, use the `ui:menuList` component, and include any of these list item components that can trigger a client-side controller action:

- `ui:actionMenuItem` - A menu item
- `ui:checkboxMenuItem` - A checkbox that supports multiple selections
- `ui:radioMenuItem` - A radio item that supports single selection

To include a separator for these menu items, use `ui:menuItemSeparator`.

This example shows several ways to create a menu.

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

```
<div style="margin:20px;">
    <div style="display:inline-block;width:50%;vertical-align:top;">
        ui:actionMenuItem
        <ui:menu>
            <ui:menuTriggerLink aura:id="trigger" label="Select your favorite team"/>
             <ui:menuList class="actionMenu" aura:id="actionMenu">
               <ui:actionMenuItem class="actionItem1" aura:id="actionItem1" label="Bayern
 Munich" click="{!c.updateTriggerLabel}"/>
                 <ui:actionMenuItem class="actionItem2" aura:id="actionItem2" label="FC
Barcelona" click="{!c.updateTriggerLabel}" disabled="true"/>
                 <ui:actionMenuItem class="actionItem3" aura:id="actionItem3" label="Inter
 Milan" click="{!c.updateTriggerLabel}"/>
                 <ui:actionMenuItem class="actionItem4" aura:id="actionItem4"
label="Manchester United" click="{!c.updateTriggerLabel}"/>
            </ui:menuList>
        </ui:menu>
     </div>

</div>
<hr/>
<p/>
<div style="margin:20px;">
    <div style="display:inline-block;width:50%;vertical-align:top;">
        ui:checkboxMenuItem
        <ui:menu>
            <ui:menuTriggerLink class="checkboxMenuLabel" aura:id="checkboxMenuLabel"
label="Select your favorite teams"/>
            <ui:menuList aura:id="checkboxMenu" class="checkboxMenu">
                <ui:checkboxMenuItem class="checkboxItem1" aura:id="checkboxItem1"
label="San Francisco 49ers"/>
                <ui:checkboxMenuItem class="checkboxItem2" aura:id="checkboxItem2"
label="Seattle Seahawks"/>
                <ui:checkboxMenuItem class="checkboxItem3" aura:id="checkboxItem3"
label="St. Louis Rams"/>
                <ui:checkboxMenuItem class="checkboxItem4" aura:id="checkboxItem4"
label="Arizona Cardinals" disabled="true" selected="true"/>
            </ui:menuList>
        </ui:menu>
        <p><ui:button class="checkboxButton" aura:id="checkboxButton"
press="{!c.getMenuSelected}" label="Check the selected menu items"/></p>
         <p><ui:outputText class="result" aura:id="result" value="Which items get
selected"/></p>
    </div>
</div>
<hr/>
<p/>
<div style="margin:20px;">
    <div style="display:inline-block;width:50%;vertical-align:top;">
        ui:radioMenuItem
        <ui:menu>
            <ui:menuTriggerLink class="radioMenuLabel" aura:id="radioMenuLabel"
label="Select a team"/>
            <ui:menuList class="radioMenu" aura:id="radioMenu">
                <ui:radioMenuItem class="radioItem1" aura:id="radioItem1" label="San
```

```
Francisco"/>
                    <ui:radioMenuItem class="radioItem2" aura:id="radioItem2" label="LA
Dodgers"/>
                  <ui:radioMenuItem class="radioItem3" aura:id="radioItem3" label="Arizona"/>

                    <ui:radioMenuItem class="radioItem4" aura:id="radioItem4" label="Diego"
disabled="true"/>
                  <ui:radioMenuItem class="radioItem5" aura:id="radioItem5" label="Colorado"/>

                </ui:menuList>
            </ui:menu>
          <p><ui:button class="radioButton" aura:id="radioButton"
press="{!c.getRadioMenuSelected}" label="Check the selected menu items"/></p>
            <p><ui:outputText class="radioResult" aura:id="radioResult" value="Which items
get selected"/> </p>
          </div>
 </div>
 <hr/>
 <p/>
 <div style="margin:20px;">
      <div style="display:inline-block;width:50%;vertical-align:top;">
          Combination menu items
          <ui:menu>
              <ui:menuTriggerLink aura:id="mytrigger" label="Select teams"/>
              <ui:menuList>
                  <ui:actionMenuItem label="Bayern Munich" click="{!c.updateLabel}"/>
                  <ui:actionMenuItem label="FC Barcelona" click="{!c.updateLabel}"/>
                  <ui:actionMenuItem label="Inter Milan" click="{!c.updateLabel}"/>
                  <ui:actionMenuItem label="Manchester United" click="{!c.updateLabel}"/>
                  <ui:menuItemSeparator/>
                  <ui:checkboxMenuItem label="San Francisco 49ers"/>
                  <ui:checkboxMenuItem label="Seattle Seahawks"/>
                  <ui:checkboxMenuItem label="St. Louis Rams"/>
                  <ui:checkboxMenuItem label="Arizona Cardinals"/>
                  <ui:menuItemSeparator/>
                  <ui:radioMenuItem label="San Francisco"/>
                  <ui:radioMenuItem label="LA Dodgers"/>
                  <ui:radioMenuItem label="Arizona"/>
                  <ui:radioMenuItem label="San Diego"/>
                  <ui:radioMenuItem label="Colorado"/>
              </ui:menuList>
          </ui:menu>
        </div>
      </div>
 </div>
</aura:component>
```

```
({
    updateTriggerLabel: function(cmp, event) {
        var triggerCmp = cmp.find("trigger");
        if (triggerCmp) {
            var source = event.getSource();
            var label = source.get("v.label");
            triggerCmp.set("v.label", label);
```

242

```
        }
    },
    updateLabel: function(cmp, event) {
        var triggerCmp = cmp.find("mytrigger");
        if (triggerCmp) {
            var source = event.getSource();
            var label = source.get("v.label");
            triggerCmp.set("v.label", label);
        }
    },
    getMenuSelected: function(cmp, event) {
        var menuCmp = cmp.find("checkboxMenu");
        var menuItems = menuCmp.get("v.childMenuItems");
        var values = [];
        for (var i = 0; i < menuItems.length; i++) {
            var c = menuItems[i];
            if (c.get("v.selected") === true) {
                values.push(c.get("v.label"));
            }
        }
        var resultCmp = cmp.find("result");
        resultCmp.set("v.value", values.join(","));
    },
    getRadioMenuSelected: function(cmp, event) {
        var menuCmp = cmp.find("radioMenu");
        var menuItems = menuCmp.get("v.childMenuItems");
        var values = [];
        for (var i = 0; i < menuItems.length; i++) {
            var c = menuItems[i];
            if (c.get("v.selected") === true) {
                values.push(c.get("v.label"));
            }
        }
        var resultCmp = cmp.find("radioResult");
        resultCmp.set("v.value", values.join(","));
    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

ui:menuItem

A UI menu item in a ui:menuList component.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
hideMenuAfterSelected	Boolean	Set to true to hide menu after the menu item is selected.	
label	String	The text displayed on the component.	
selected	Boolean	The status of the menu item. True means this menu item is selected; False is not selected.	
type	String	The concrete type of the menu item. Accepted values are 'action', 'checkbox', 'radio', 'separator' or any namespaced component descriptor, e.g. ns:xxxxmenuItem.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.

Event Name	Event Type	Description
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.

ui:menuItemSeparator

A menu separator to divide menu items, such as ui:radioMenuItem, and used in a ui:menuList component.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.

Event Name	Event Type	Description
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

ui:menuList

A menu component that can contain menu items: ui:actionMenuItem, ui:checkboxMenuItem, and ui:radioMenuItem, and a separator component ui:menuItemSeparator. This component can be nested in a ui:menu component and can be used together with a ui:menuTriggerLink component, which triggers the menu items.

Attributes

Attribute Name	Attribute Type	Description	Required?
autoPosition	Boolean	Move the popup target up when there is not enough space at the bottom to display. Note: even if autoPosition is set to false, popup will still position the menu relative to the trigger. To override default positioning, use manualPosition attribute.	
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
closeOnClickOutside	Boolean	Close target when user clicks or taps outside of the target	
closeOnTabKey	Boolean	Indicates whether to close the target list on tab key or not.	
curtain	Boolean	Whether or not to apply an overlay under the target.	
menuItems	List	A list of menu items set explicitly using instances of the Java class: aura. components.ui.MenuItem.	
visible	Boolean	Controls the visibility of the menu. The default is false, which hides the menu.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.

246

Event Name	Event Type	Description
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
collapse	COMPONENT	Indicates that a component collapses.
expand	COMPONENT	Indicates that a component expands.
menuSelect	COMPONENT	Indicates that the user selects a menu item inside a menu component.

ui:menuTrigger

A trigger that expands and collapses a menu, used in a ui:menu component.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
title	String	The text to display as a tooltip when the mouse pointer hovers over this component.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.

Event Name	Event Type	Description
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
menuTriggerPress	COMPONENT	Indicates that the menu trigger is clicked.

ui:menuTriggerLink

A link that triggers a dropdown menu.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
label	String	The text displayed on the component.	
title	String	The text to display as a tooltip when the mouse pointer hovers over this component.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.

Event Name	Event Type	Description
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.
menuTriggerPress	COMPONENT	Indicates that the menu trigger is clicked.

`ui:message`

Represents a message of varying severity levels

The `severity` attribute indicates a message's severity level and determines the style to use when displaying the message. If the `closable` attribute is set to true, the message can be dismissed by pressing the × symbol.

This example shows a success message when a condition evaluates to true or an error message otherwise.

```
<aura:component>
    <aura:attribute name="myBool" type="Boolean" default="true"/>

    <aura:renderIf isTrue="!{v.myBool}">
        <ui:message title="Success" severity="confirm">
           The operation is successful.
        </ui:message>
        <aura:set attribute="else">
            <ui:message title="Error" severity="error">
                This is an error.
            </ui:message>
        </aura:set>
    </aura:renderIf>
</aura:component>
```

This example shows messages in varying severity levels.

```
<aura:component access="global">
 <ui:message title="Confirmation" severity="confirm" closable="true">
   This is a confirmation message.
 </ui:message>
 <ui:message title="Information" severity="info" closable="true">
   This is a message.
 </ui:message>
 <ui:message title="Warning" severity="warning" closable="true">
   This is a warning.
 </ui:message>
 <ui:message title="Error" severity="error" closable="true">
   This is an error message.
 </ui:message>

</aura:component>
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
closable	Boolean	Specifies whether to display an 'x' that will close the alert when clicked. Default value is 'false'.	
severity	String	The severity of the message. Possible values: message (default), confirm, info, warning, error	
title	String	The title text for the message.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

ui:outputCheckbox

Displays a checkbox in a checked or unchecked state.

A `ui:outputCheckbox` component represents a checkbox that is rendered as an HTML `img` tag. This component can be used with `ui:inputCheckbox`, which enables users to select or deselect the checkbox. To select or deselect the checkbox, set the `value` attribute to `true` or `false`. To display a checkbox, you can use an attribute value and bind it to the `ui:outputCheckbox` component.

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

The previous example renders the following HTML.

```
<img class="checked uiImage uiOutputCheckbox" alt="checkbox checked" src="path/to/checkbox">
```

This example shows how you can use the `ui:inputCheckbox` component.

```
<aura:component>
 <aura:attribute name="myBool" type="Boolean" default="true"/>
 <ui:inputCheckbox aura:id="checkbox" label="Select?" change="{!c.onCheck}"/>
 <p>Selected:</p>
 <p><ui:outputText class="result" aura:id="checkResult" value="false" /></p>
 <p>The following checkbox uses a component attribute to bind its value.</p>
 <ui:outputCheckbox aura:id="output" value="{!v.myBool}"/>
</aura:component>
```

```
({
  onCheck: function(cmp, evt) {
   var checkCmp = cmp.find("checkbox");
   resultCmp = cmp.find("checkResult");
   resultCmp.set("v.value", ""+checkCmp.get("v.value"));

  }
}
```

Attributes

Attribute Name	Attribute Type	Description	Required?
altChecked	String	The alternate text description when the checkbox is checked. Default value is "checkbox checked".	
altUnchecked	String	The alternate text description when the checkbox is unchecked. Default value is "checkbox unchecked".	
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
value	Boolean	Specifies whether the checkbox is checked.	Yes

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.

Event Name	Event Type	Description
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

ui:outputCurrency

Displays the currency in the default or specified format, such as with specific currency code or decimal places.

A `ui:outputCurrency` component represents a number as a currency that is wrapped in an HTML `span` tag. This component can be used with `ui:inputCurrency`, which takes in a number as a currency. To display a currency, you can use an attribute value and bind it to the `ui:outputCurrency` component.

```
<aura:attribute name="myCurr" type="Decimal" default="50000"/>
<ui:outputCurrency aura:id="curr" value="{!v.myCurr}"/>
```

The previous example renders the following HTML.

```
<span class="uiOutputCurrency">$50,000.00</span>
```

To override the browser's locale, use the `currencySymbol` attribute.

```
<aura:attribute name="myCurr" type="Decimal" default="50" currencySymbol="£"/>
```

You can also override it by specifying the format.

```
var curr = cmp.find("curr");
curr.set("v.format", '£#,###.00');
```

This example shows how you can bind data from a `ui:inputCurrency` component.

```
<aura:component>
    <ui:inputCurrency aura:id="amount" label="Amount" class="field" value="50"/>
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
  You entered: <ui:outputCurrency aura:id="oCurrency" value=""/>
 </div>
</aura:component>
```

```
({
  setOutput : function(component, event, helper) {
  var el = component.find("msg");
    $A.util.removeClass(el.getElement(), 'hide');

      var amount = component.find("amount").get("v.value");
      var oCurrency = component.find("oCurrency");
      oCurrency.set("v.value", amount);
  }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
currencyCode	String	The ISO 4217 currency code specified as a String, e.g. "USD".	
currencySymbol	String	The currency symbol specified as a String.	
format	String	The format of the number. For example, format=".00" displays the number followed by two decimal places. If not specified, the default format based on the browser's locale will be used.	
value	BigDecimal	The output value of the currency, which is defined as type Decimal.	Yes

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

`ui:outputDate`

Displays a date in the default or specified format based on the user's locale.

A `ui:outputDate` component represents a date output in the YYYY-MM-DD format and is wrapped in an HTML `span` tag. This component can be used with `ui:inputDate`, which takes in a date input. `ui:outputDate` retrieves the browser's locale information and displays the date accordingly. To display a date, you can use an attribute value and bind it to the `ui:outputDate` component.

```
<aura:attribute name="myDate" type="Date" default="2014-09-29"/>
<ui:outputDate value="{!v.myDate}"/>
```

The previous example renders the following HTML.

```
<span class="uiOutputDate">Sep 29, 2014</span>
```

This example shows how you can bind data from the `ui:inputDate` component.

```
<aura:component>
 <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
 <aura:attribute name="today" type="Date" default=""/>

    <ui:inputDate aura:id="expdate" label="Today's Date" class="field" value="{!v.today}"
 displayDatePicker="true" />
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
  You entered: <ui:outputDate aura:id="oDate" value="" />
 </div>
</aura:component>
```

```
({
    doInit : function(component, event, helper) {
        var today = new Date();
        component.set('v.today', today.getFullYear() + "-" + (today.getMonth() + 1) + "-"
 + today.getDate());
        component.set('v.deadline', today);
    },

    setOutput : function(component, event, helper) {
     var el = component.find("msg");
     $A.util.removeClass(el.getElement(), 'hide');
        var expdate = component.find("expdate").get("v.value");

        var oDate = component.find("oDate");
        oDate.set("v.value", expdate);

    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
format	String	A string (pattern letters are defined in java.text.SimpleDateFormat) used to format the date and time of the value attribute.	
langLocale	String	The language locale used to format date value.	
value	String	The output value of the date. It should be a date string in ISO-8601 format (YYYY-MM-DD).	Yes

254

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

`ui:outputDateTime`

Displays a date, time in a specified or default format based on the user's locale.

A `ui:outputDateTime` component represents a date and time output that is wrapped in an HTML `span` tag. This component can be used with `ui:inputDateTime`, which takes in a date input. `ui:outputDateTime` retrieves the browser's locale information and displays the date accordingly. To display a date and time, you can use an attribute value and bind it to the `ui:outputDateTime` component.

```
<aura:attribute name="myDateTime" type="Date" default="2014-09-29T00:17:08z"/>
<ui:outputDateTime value="{!v.myDateTime}"/>
```

The previous example renders the following HTML.

```
<span class="uiOutputDateTime">Sep 29, 2014 12:17:08 AM</span>
```

This example shows how you can bind data from a `ui:inputDateTime` component.

```
<aura:component>
 <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
 <aura:attribute name="today" type="Date" default=""/>

    <ui:inputDateTime aura:id="today" label="Time" class="field" value=""
displayDatePicker="true" />
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

    <div aura:id="msg" class="hide">
  You entered: <ui:outputDateTime aura:id="oDateTime" value=""  />
 </div>
</aura:component>
```

```
({
    doInit : function(component, event, helper) {
        var today = new Date();
        component.set('v.today', today.getFullYear() + "-" + (today.getMonth() + 1) + "-"
 + today.getDate());
```

```
    },

    setOutput : function(component, event, helper) {
     var el = component.find("msg");
     $A.util.removeClass(el.getElement(), 'hide');

         var todayVal = component.find("today").get("v.value");
         var oDateTime = component.find("oDateTime");
         oDateTime.set("v.value", todayVal);


    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
format	String	A string (pattern letters are defined in java.text.SimpleDateFormat) used to format the date and time of the value attribute.	
langLocale	String	The language locale used to format date value.	
timezone	String	The timezone ID, for example, America/Los_Angeles.	
value	String	An ISO8601-formatted string representing a date time.	Yes

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

256

ui:outputEmail

Displays an email address in an HTML anchor (<a>) element. The leading and trailing space are trimmed.

A `ui:outputEmail` component represents an email output that is wrapped in an HTML `span` tag. This component can be used with `ui:inputEmail`, which takes in an email input. The email output is wrapped in an HTML anchor element and `mailto` is automatically appended to it. This is a simple set up of a `ui:outputEmail` component.

```
<ui:outputEmail value="abc@email.com"/>
```

The previous example renders the following HTML.

```
<span><a href="mailto:abc@email.com" class="uiOutputEmail">abc@email.com</a></span>
```

This example shows how you can bind data from a `ui:inputEmail` component.

```
<aura:component>
    <ui:inputEmail aura:id="email" label="Email" class="field" value="manager@email.com"/>

    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
     You entered: <ui:outputEmail aura:id="oEmail" value="Email" />
 </div>

</aura:component>
```

```
({
    setOutput : function(component, event, helper) {
     var el = component.find("msg");
     $A.util.removeClass(el.getElement(), 'hide');

        var email = component.find("email").get("v.value");
        var oEmail = component.find("oEmail");
        oEmail.set("v.value", email);

    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
value	String	The output value of the email	Yes

257

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

`ui:outputNumber`

Displays the number in the default or specified format. Supports up to 18 digits before the decimal place.

A `ui:outputNumber` component represents a number output that is rendered as an HTML `span` tag. This component can be used with `ui:inputNumber`, which takes in a number input. `ui:outputNumber` retrieves the browser's locale information and displays the number in the given decimal format. To display a number, you can use an attribute value and bind it to the `ui:outputNumber` component.

```
<aura:attribute name="myNum" type="Decimal" default="10.10"/>
<ui:outputNumber value="{!v.myNum}" format=".00"/>
```

The previous example renders the following HTML.

```
<span class="uiOutputNumber">10.10</span>
```

This example retrieves the value of a `ui:intputNumber` component, validates the input, and displays it using `ui:outputNumber`.

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

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

        var myOutput = component.find("outNum");

        myOutput.set("v.value", value);

        // Check if input is numeric
        if (isNaN(value)) {
```

```
            // Set error message
        inputCmp.setValid("v.value", false);
        inputCmp.addErrors("v.value", [{message:"Input not a number: " + value}]);
    } else {
        // Clear error
        inputCmp.setValid("v.value", true);
    }
    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
format	String	The format of the number. For example, format=".00" displays the number followed by two decimal places. If not specified, the Locale default format will be used.	
value	BigDecimal	The number displayed when this component is rendered.	Yes

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

ui:outputPhone

Displays the phone number in a URL link format.

A `ui:outputPhone` component represents a phone number output that is wrapped in an HTML `span` tag. This component can be used with `ui:inputPhone`, which takes in a phone number input. The following example is a simple set up of a `ui:outputPhone` component.

```
<ui:outputPhone value="415-123-4567"/>
```

The previous example renders the following HTML.

```
<span class="uiOutputPhone">415-123-4567</span>
```

When viewed on a mobile device, the example renders as an actionable link.

```
<span class="uiOutputPhone">
    <a href="tel:415-123-4567">415-123-4567</a>
</span>
```

This example shows how you can bind data from a `ui:inputPhone` component.

```
<aura:component>
    <ui:inputPhone aura:id="phone" label="Phone Number" class="field" value="415-123-4567"
 />
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
  You entered: <ui:outputPhone aura:id="oPhone" value="" />
 </div>
</aura:component>
```

```
({

    setOutput : function(component, event, helper) {
     var el = component.find("msg");
     $A.util.removeClass(el.getElement(), 'hide');

        var phone = component.find("phone").get("v.value");
        var oPhone = component.find("oPhone");
        oPhone.set("v.value", phone);
    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
value	String	The phone number displayed when this component is rendered.	Yes

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

ui:outputRichText

Displays richly-formatted text including tags such as paragraph, image, and hyperlink, as specified in the value attribute.

A `ui:outputRichText` component represents rich text and can be used to display input from a `ui:inputRichText` component.

For example, you can enter bold or colored text via a `ui:inputRichText` component and bind its value to a `ui:outputRichText` component, which results in the following HTML.

```
<div class="uiOutputRichText">
    <b>Aura</b>, <span style="color:red">input rich text demo</span>
</div>
```

This example shows how you can bind data from a `ui:inputRichText` component.

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

```
({
 getInput : function(cmp) {

  var userInput = cmp.find("inputRT").get("v.value");
  var output = cmp.find("outputRT");
  output.set("v.value", userInput);
 }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

`ui:outputText`

Displays text as specified by the value attribute.

A `ui:outputText` component represents text output that is wrapped in an HTML `span` tag. This component can be used with `ui:inputText`, which takes in a text input. To display text, you can use an attribute value and bind it to the `ui:outputText` component.

```
<aura:attribute name="myText" type="String" default="some string"/>
<ui:outputText value="{!v.myText}" label="my output"/>
```

The previous example renders the following HTML.

```
<span dir="ltr" class="uiOutputText">
    some string
</span>
```

This example shows how you can bind data from an `ui:inputText` component.

```
<aura:component>
 <ui:inputText aura:id="name" label="Enter some text" class="field" value="My Text"/>
    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

 <div aura:id="msg" class="hide">
  You entered: <ui:outputText aura:id="oName" value=""/>
```

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

```
({
  setOutput : function(component, event, helper) {
  var el = component.find("msg");
     $A.util.removeClass(el.getElement(), 'hide');

        var name = component.find("name").get("v.value");
        var oName = component.find("oName");
        oName.set("v.value", name);
  }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
value	String	The text displayed when this component is rendered.	Yes

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

ui:outputTextArea

Displays the text area as specified by the value attribute.

A `ui:outputTextArea` component represents text output that is wrapped in an HTML `span` tag. This component can be used with `ui:inputTextArea`, which takes in a multi-line text input. To display text, you can use an attribute value and bind it to the `ui:outputTextArea` component.

```
<aura:attribute name="myTextArea" type="String" default="some string"/>
<ui:outputTextArea value="{!v.myTextArea}"/>
```

The previous example renders the following HTML.

```
<span class="uiOutputTextArea">some string</span>
```

This example shows how you can bind data from the `ui:inputTextArea` component.

```
<aura:component>
    <ui:inputTextArea aura:id="comments" label="Comments"  value="My comments" rows="5"/>

    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>

    <div aura:id="msg" class="hide">
  You entered: <ui:outputTextArea aura:id="oTextarea" value=""/>
 </div>
</aura:component>
```

```
({
    setOutput : function(component, event, helper) {
     var el = component.find("msg");
     $A.util.removeClass(el.getElement(), 'hide');

        var comments = component.find("comments").get("v.value");
        var oTextarea = component.find("oTextarea");
        oTextarea.set("v.value", comments);
    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
value	String	The text to display.	Yes

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.

Event Name	Event Type	Description
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

`ui:outputURL`

Displays a link to a URL as specified by the value attribute, rendered on a given text (label attribute) and image, if any.

A `ui:outputURL` component represents a URL that is wrapped in an HTML `a` tag. This component can be used with `ui:inputURL`, which takes in a URL input. To display a URL, you can use an attribute value and bind it to the `ui:outputURL` component.

```
<aura:attribute name="myURL" type="String" default="http://www.google.com"/>
<ui:outputURL value="{!v.myURL}" label="{!v.myURL}"/>
```

The previous example renders the following HTML.

```
<a href="http://www.google.com" dir="ltr" class="uiOutputURL">http://www.google.com</a>
```

This example shows how you can bind data from a `ui:inputURL` component.

```
<aura:component>
    <ui:inputURL aura:id="url" label="Venue URL" class="field" value="http://www.myURL.com"/>

    <ui:button class="btn" label="Submit" press="{!c.setOutput}"/>
 <div aura:id="msg" class="hide">
  You entered: <ui:outputURL aura:id="oURL" value=""/>
 </div>
</aura:component>
```

```
({
    setOutput : function(component, event, helper) {
     var el = component.find("msg");
     $A.util.removeClass(el.getElement(), 'hide');

        var url = component.find("url").get("v.value");
        var oURL = component.find("oURL");
        oURL.set("v.value", url);
        oURL.set("v.label", url);
    }
})
```

265

Attributes

Attribute Name	Attribute Type	Description	Required?
alt	String	The alternate text description for image (used when there is no label)	
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
iconClass	String	The CSS style used to display the icon or image.	
label	String	The text displayed on the component.	
target	String	The target destination where this rendered component is displayed. Possible values: _blank, _parent, _self, _top	
title	String	The text to display as a tooltip when the mouse pointer hovers over this component.	
value	String	The text displayed when this component is rendered.	Yes

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.

ui:radioMenuItem

A menu item with a radio button that indicates a mutually exclusive selection and can be used to invoke an action. This component is nested in a ui:menu component.

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
class	String	A CSS style to be attached to the component. This style is added in addition to base styles output by the component.	
disabled	Boolean	Specifies whether the component should be displayed in a disabled state. Default value is "false".	
hideMenuAfterSelected	Boolean	Set to true to hide menu after the menu item is selected.	
label	String	The text displayed on the component.	
selected	Boolean	The status of the menu item. True means this menu item is selected; False is not selected.	
type	String	The concrete type of the menu item. Accepted values are 'action', 'checkbox', 'radio', 'separator' or any namespaced component descriptor, e.g. ns:xxxxmenuItem.	

Events

Event Name	Event Type	Description
mouseup	COMPONENT	Indicates that the user has released the mouse button.
mousedown	COMPONENT	Indicates that the user has pressed a mouse key.
mousemove	COMPONENT	Indicates that the user has moved the mouse pointer.
dblclick	COMPONENT	Indicates that a component has been double-clicked.
mouseout	COMPONENT	Indicates that the user has moved the mouse pointer away from the component.
click	COMPONENT	Indicates that a component has been clicked.
mouseover	COMPONENT	Indicates that the user has moved the mouse pointer over the component.
keyup	COMPONENT	Indicates that the user has released a keyboard key.
keypress	COMPONENT	Indicates that the user has pressed and held down a keyboard key.
select	COMPONENT	Indicates that the user has made a selection.
keydown	COMPONENT	Indicates that the user has pressed and released a keyboard key.
focus	COMPONENT	Indicates that a component has been put on focus.
blur	COMPONENT	Indicates that a component has been put out of focus.

`ui:spinner`

A loading spinner to be used while the real component body is being loaded

To toggle the spinner, use `get("e.toggle")`, set the `isVisible` parameter to `true` or `false`, and then fire the event.

This example shows a spinner when a component is expecting a server response and removes the spinner when the component is no longer waiting for a response.

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

   <center><ui:spinner aura:id="spinner"/></center>
</aura:component>
```

This client-side controllers shows and hides the spinner accordingly.

```
({
     showSpinner : function (component, event, helper) {
        var spinner = component.find('spinner');
        var evt = spinner.get("e.toggle");
        evt.setParams({ isVisible : true });
        evt.fire();
    },

    hideSpinner : function (component, event, helper) {
        var spinner = component.find('spinner');
        var evt = spinner.get("e.toggle");
        evt.setParams({ isVisible : false });
        evt.fire();
    }
})
```

This example shows a spinner that can be toggled.

```
<aura:component access="global">
 <ui:spinner aura:id="spinner"/>
 <ui:button press="{!c.toggleSpinner}" label="Toggle Spinner" />
</aura:component>
```

```
({
    toggleSpinner: function(cmp, event) {
        var spinner = cmp.find('spinner');
        var evt = spinner.get("e.toggle");

    if(!$A.util.hasClass(spinner.getElement(), 'hideEl')){
        evt.setParams({ isVisible : false });
     }
    else {
        evt.setParams({ isVisible : true });
 }
    evt.fire();
    }
})
```

Attributes

Attribute Name	Attribute Type	Description	Required?
body	Component[]	The body of the component. In markup, this is everything in the body of the tag.	
isVisible	Boolean	Specifies whether or not this spinner should be visible. Defaults to true.	

Events

Event Name	Event Type	Description
toggleLoadingIndicator	COMPONENT	Change the visibility of a ui:spinner component.

Event Reference

Use out-of-the-box events to enable component interaction within Salesforce1 or within your Lightning components. For example, these events enable your components to open a record create or edit page, or navigate to a record.

force:createRecord

Opens the page to create a new record for the specified `entityApiName`, for example, "Account" or "myNamespace__MyObject__c".

To display the record create page for an object, set the object name on the `entityApiName` parameter and fire the event. `recordTypeId` is optional and, if provided, specifies the record type for the created object. This example displays the record create panel for contacts.

```
createRecord : function (component, event, helper) {
    var createRecordEvent = $A.get("e.force:createRecord");
    createRecordEvent.setParams({
        "entityApiName": "Contact"
    });
    createRecordEvent.fire();
}
```

📝 **Note:** This event is supported in Salesforce1 only. If used outside of Salesforce1, this event won't be handled properly.

Attribute Name	Type	Description
entityApiName	String	Required. The API name of the custom or standard object, such as "Account", "Case", "Contact", "Lead", "Opportunity", or "namespace__objectName__c".
recordTypeId	String	The ID of the record type, if record types are available for the object.

force:editRecord

Opens the page to edit the record specified by `recordId`.

To display the record edit page for an object, set the object name on the `recordId` attribute and fire the event. This example displays the record edit page for a contact that's specified by `recordId`.

```
editRecord : function(component, event, helper) {
    var editRecordEvent = $A.get("e.force:editRecord");
    editRecordEvent.setParams({
        "recordId": component.get("v.contact.Id")
    });
    editRecordEvent.fire();
}
```

📝 **Note:** This event is supported in Salesforce1 only. If used outside of Salesforce1, this event won't be handled properly.

Attribute Name	Type	Description
recordId	String	Required. The record ID associated with the record to be edited.

force:navigateToList

Navigates to the list view specified by `listViewId`.

To navigate to a list view, set the list view ID on the `listViewId` attribute and fire the event. This example displays the list views for contacts.

```
gotoList : function (component, event, helper) {
    var action = component.get("c.getListViews");
    action.setCallback(this, function(response){
        var state = response.getState();
        if (state === "SUCCESS") {
            var listviews = response.getReturnValue();
            var navEvent = $A.get("e.force:navigateToList");
            navEvent.setParams({
                "listViewId": listviews.Id,
                "listViewName": null,
                "scope": "Contact"
            });
            navEvent.fire();
        }
    });
    $A.enqueueAction(action);
}
```

This Apex controller returns all list views for the contact object.

```
@AuraEnabled
public static List<ListView> getListViews() {
    List<ListView> listviews =
        [SELECT Id, Name FROM ListView WHERE SobjectType = 'Contact'];

    // Perform isAccessible() check here
    return listviews;
}
```

270

You can also provide a single list view ID by providing the list view name you want to navigate to in the SOQL query.

```
SELECT Id, Name FROM ListView WHERE SobjectType = 'Contact' and Name='All Contacts'
```

📝 Note: This event is supported in Salesforce1 only. If used outside of Salesforce1, this event won't be handled properly.

Attribute Name	Type	Description
listViewId	String	Required. The ID of the list view to be displayed.
listViewName	String	Specifies the name for the list view and doesn't need to match the actual name. To use the actual name that's saved for the list view, set listViewName to null.
scope	String	The name of the sObject in the view, for example, "Account" or "namespace__MyObject__c".

SEE ALSO:

[CRUD and Field-Level Security (FLS)](#)

force:navigateToObjectHome

Navigates to the object home specified by the scope attribute.

To navigate to an object home, set the object name on the scope attribute and fire the event. This example displays the home page for a custom object.

```
navHome : function (component, event, helper) {
    var homeEvent = $A.get("e.force:navigateToObjectHome");
    homeEvent.setParams({
        "scope": "myNamespace__myObject__c"
    });
    homeEvent.fire();
}
```

📝 Note: This event is supported in Salesforce1 only. If used outside of Salesforce1, this event won't be handled properly.

Attribute Name	Type	Description
scope	String	Required. The API name of the custom or standard object, such as "Contact", or "namespace__objectName__c".
resetHistory	Boolean	Resets history if set to true. Defaults to false, which provides a Back button in Salesforce1.

force:navigateToRelatedList

Navigates to the related list specified by parentRecordId.

To navigate to a related list, set the parent record ID on the `parentRecordId` attribute and fire the event. For example, to display a related list for a Contact object, the `parentRecordId` is `Contact.Id`. This example displays the related cases for a contact record.

```
gotoRelatedList : function (component, event, helper) {
    var relatedListEvent = $A.get("e.force:navigateToRelatedList");
    relatedListEvent.setParams({
        "relatedListId": "Cases",
        "parentRecordId": component.get("v.contact.Id")
    });
    relatedListEvent.fire();
}
```

📝 Note: This event is supported in Salesforce1 only. If used outside of Salesforce1, this event won't be handled properly.

Attribute Name	Type	Description
parentRecordId	String	Required. The ID of the parent record.
relatedListId	String	Required. The API name of the related list to display, such as "Contacts" or "Opportunities".

force:navigateToSObject

Navigates to an sObject record specified by `recordId`.

To display the record view, set the record ID on the `recordId` attribute and fire the event. The record view contains slides that displays the Chatter feed, the record details, and related information. This example displays the related information slide of a record view for the specified record ID.

```
createRecord : function (component, event, helper) {
    var navEvt = $A.get("e.force:navigateToSObject");
    navEvt.setParams({
      "recordId": "00QB0000000ybNX",
      "slideDevName": "related"
    });
    navEvt.fire();
}
```

📝 Note: This event is supported in Salesforce1 only. If used outside of Salesforce1, this event won't be handled properly.

Attribute Name	Type	Description
recordId	String	Required. The record ID.
slideDevName	String	Specifies the slide within the record view to display initially. Valid options are: • `detail`: The record detail slide. This is the default value. • `chatter`: The Chatter slide • `related`: The related information slide

force:navigateToURL

Navigates to the specified URL.

Relative and absolute URLs are supported. Relative URLs are relative to the Salesforce1 mobile browser app domain, and retain navigation history. External URLs open in a separate browser window.

Use relative URLs to navigate to different screens within your app. Use external URLs to allow the user to access a different site or app, where they can take actions that don't need to be preserved in your app. To return to your app, the separate window that's opened by an external URL must be closed when the user is finished with the other app. The new window has a separate history from your app, and this history is discarded when the window is closed. This also means that the user can't click a Back button to go back to your app; the user must close the new window.

`mailto:`, `tel:`, `geo:`, and other URL schemes are supported for launching external apps and attempt to "do the right thing." However, support varies by mobile platform and device. `mailto:` and `tel:` are reliable, but we recommend that you test any other URLs on a range of expected devices.

> **Note:** Only standard URL schemes are supported by `navigateToURL`. To access custom schemes, use `window.location` instead.

When using `mailto:` and `tel:` URL schemes, you can also consider using `ui:outputEmail` and `ui:outputURL` components.

This example navigates a user to the opportunity page, `/006/o`, using a relative URL.

```
gotoURL : function (component, event, helper) {
    var urlEvent = $A.get("e.force:navigateToURL");
    urlEvent.setParams({
      "url": "/006/o"
    });
    urlEvent.fire();
}
```

This example opens an external website when the link is clicked.

```
navigate : function(component, event, helper) {

    //Find the text value of the component with aura:id set to "address"
    var address = component.find("address").get("v.value");

    var urlEvent = $A.get("e.force:navigateToURL");
    urlEvent.setParams({
      "url": 'https://www.google.com/maps/place/' + address
    });
    urlEvent.fire();
}
```

> **Note:** This event is supported in Salesforce1 only. If used outside of Salesforce1, this event won't be handled properly.

Attribute Name	Type	Description
isredirect	Boolean	Indicates that the new URL should replace the current one in the navigation history. Defaults to `false`.
url	String	Required. The URL of the target.

force:recordSave

Saves a record.

`force:recordSave` is handled by the `force:recordEdit` component. This examples shows a `force:recordEdit` component, which takes in user input to update a record specified by the `recordId` attribute. The button fires the `force:recordSave` event.

```
<force:recordEdit aura:id="edit" recordId="a02D0000006V8Ni"/>
<ui:button label="Save" press="{!c.save}"/>
```

This client-side controller fires the event to save the record.

```
save : function(component, event, helper) {
    component.find("edit").get("e.recordSave").fire();
    // Update the component
    helper.getRecords(component);
}
```

> Note: This event is supported in Salesforce1 only. If used outside of Salesforce1, this event won't be handled properly.

force:recordSaveSuccess

Indicates that the record has been successfully saved.

`force:recordSaveSuccess` is handled by the `force:recordEdit` component. This examples shows a `force:recordEdit` component, which takes in user input to update a record specified by the `recordId` attribute. The button fires the `force:recordSave` event.

```
<force:recordEdit aura:id="edit" recordId="a02D0000006V8Ni" onSaveSuccess="{!c.onSuccess}"/>
<ui:button label="Save" press="{!c.save}"/>
```

This client-side controller fires the event to save the record.

```
save : function(component, event, helper) {
    component.find("edit").get("e.recordSave").fire();
}
```

This client-side controller handles the `onSaveSuccess` event by firing another event to refresh the component.

```
onSuccess: function(component) {
    var refresh = $A.get("e.force:refreshView");
    refresh.fire();
}
```

> Note: This event is supported in Salesforce1 only. If used outside of Salesforce1, this event won't be handled properly.

force:refreshView

Reloads the view.

To refresh a view, run `$A.get("e.force:refreshView").fire();`, which reloads all data for the view.

274

This example refreshes the view after an action is successfully completed.

```
refresh : function(component, event, helper) {
    var action = cmp.get('c.myController');
    action.setCallback(cmp,
        function(response) {
            var state = response.getState();
            if (state === 'SUCCESS'){
                $A.get('e.force:refreshView').fire();
            } else {
                //do something
            }
        }
    );
    $A.enqueueAction(action);
}
```

📝 Note: This event is supported in Salesforce1 only. If used outside of Salesforce1, this event won't be handled properly.

force:showToast

Displays a message in a popup.

A toast displays a message below the header at the top of a view. The message is specified by the `message` attribute.

This example displays a toast message **Success!** `The record has been updated successfully.`.

```
showToast : function(component, event, helper) {
    var toastEvent = $A.get("e.force:showToast");
    toastEvent.setParams({
        "title": "Success!",
        "message": "The record has been updated successfully."
    });
    toastEvent.fire();
}
```

📝 Note: This event is supported in Salesforce1 only. If used outside of Salesforce1, this event won't be handled properly.

Attribute Name	Type	Description
title	String	Specifies the title in bold for the toast.
message	String	Specifies the message to display.
key	String	Specifies an icon type.
duration	Integer	Toast duration in milliseconds. Default is 3000ms.

lntg:selectSObject

Sends the `recordId` of an object when it's selected in the UI.

To select an object, set the record ID on the `recordId` attribute. Optionally, specify a channel for this event so that your components can select if they want to listen to particular event messages.

```
selectedObj: function(component, event) {
 var selectedObjEvent = $A.get("e.lntg:selectSObject");
 selectedObjEvent.setParams({
         "recordId": "0061a000004x8e1",
         "channel": "AccountsChannel"
 });
 selectedObj.fire();
}
```

Attribute Name	Type	Description
recordId	String	Required. The record ID associated with the record to select.
channel	String	Specify this field if you want particular components to process some event messages while ignoring others.

lntg:sendMessage

Passes a message between two components.

To send a message, specify a string of text that you want to pass between components. Optionally, specify a channel for this event so that your components can select if they want to listen to particular event messages

```
sendMsg: function(component, event) {
 var sendMsgEvent = $A.get("e.lntg:sendMessage");
 sendMsgEvent.setParams({
         "message": "Hello World",
         "channel": "AccountsChannel"
 });
 sendMsg.fire();
}
```

Attribute Name	Type	Description
message	String	Required. The text that you want to pass between components.
channel	String	Specify this field if you want particular components to process some event messages while ignoring others.

ui:clearErrors

Indicates that any validation errors should be cleared.

To set a handler for the `ui:clearErrors` event, use the `onClearErrors` system attribute on a component that extends `ui:input`, such as `ui:inputNumber`.

The following `ui:inputNumber` component handles an error when the `ui:button` component is pressed. You can fire and handle these events in a client-side controller.

```
<aura:component>
    Enter a number:
    <!-- onError calls your client-side controller to handle a validation error -->
    <!-- onClearErrors calls your client-side controller to handle clearing of errors -->

    <ui:inputNumber aura:id="inputCmp" onError="{!c.handleError}"
onClearErrors="{!c.handleClearError}"/> <br/>

    <!-- press calls your client-side controller to trigger validation errors -->
    <ui:button label="Submit" press="{!c.doAction}"/>
</aura:component>
```

For more information, see Validating Fields on page 140.

ui:collapse

Indicates that a menu component collapses.

For example, the `ui:menuList` component registers this event and handles it when it's fired.

```
<aura:registerEvent name="menuCollapse"  type="ui:collapse"
                    description="The event fired when the menu list collapses." />
```

You can handle this event in a `ui:menuList` component instance. This example shows a menu component with two list items. It handles the `ui:collapse` and `ui:expand` events.

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Contacts"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu" menuCollapse="{!c.addClass}"
 menuExpand="{!c.removeClass}">
            <ui:actionMenuItem aura:id="item1" label="All Contacts"
click="{!c.doSomething}"/>
          <ui:actionMenuItem aura:id="item2" label="All Primary" click="{!c.doSomething}"/>

        </ui:menuList>
</ui:menu>
```

This client-side controller adds a CSS class to the trigger when the menu is collapsed and removes it when the menu is expanded.

```
({
    addClass : function(component, event, helper) {
        var trigger = component.find("trigger");
        $A.util.addClass(trigger, "myClass");
    },
    removeClass : function(component, event, helper) {
        var trigger = component.find("trigger");
        $A.util.removeClass(trigger, "myClass");
    }
})
```

ui:expand

Indicates that a menu component expands.

For example, the `ui:menuList` component registers this event and handles it when it's fired.

```
<aura:registerEvent name="menuExpand"  type="ui:expand"
                    description="The event fired when the menu list displays." />
```

You can handle this event in a `ui:menuList` component instance. This example shows a menu component with two list items. It handles the `ui:collapse` and `ui:expand` events.

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Contacts"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu" menuCollapse="{!c.addClass}"
 menuExpand="{!c.removeClass}">
            <ui:actionMenuItem aura:id="item1" label="All Contacts"
click="{!c.doSomething}"/>
          <ui:actionMenuItem aura:id="item2" label="All Primary" click="{!c.doSomething}"/>

        </ui:menuList>
</ui:menu>
```

This client-side controller adds a CSS class to the trigger when the menu is collapsed and removes it when the menu is expanded.

```
({
    addClass : function(component, event, helper) {
        var trigger = component.find("trigger");
        $A.util.addClass(trigger, "myClass");
    },
    removeClass : function(component, event, helper) {
        var trigger = component.find("trigger");
        $A.util.removeClass(trigger, "myClass");
    }
})
```

ui:menuSelect

Indicates that a menu item has been selected in the menu component.

For example, the `ui:menuList` component registers this event so it can be fired by the component.

```
<aura:registerEvent name="menuSelect"  type="ui:menuSelect"
                    description="The event fired when a menu item is selected." />
```

You can handle this event in a `ui:menuList` component instance. This example shows a menu component with two list items. It handles the `ui:menuSelect` event and `click` events.

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Contacts"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu" menuSelect="{!c.selected}">
            <ui:actionMenuItem aura:id="item1" label="All Contacts"
click="{!c.doSomething}"/>
          <ui:actionMenuItem aura:id="item2" label="All Primary" click="{!c.doSomething}"/>
```

```
        </ui:menuList>
</ui:menu>
```

When a menu item is clicked, the `click` event is handled before the `ui:menuSelect` event, which corresponds to `doSomething` and `selected` client-side controllers in the following example.

```
({
    selected : function(component, event, helper) {
        var selected = event.getParam("selectedItem");

        // returns label of selected item
        var selectedLabel = selected.get("v.label");
    },

    doSomething : function(component, event, helper) {
        console.log("do something");
    }
})
```

Attribute Name	Type	Description
selectedItem	Component[]	The menu item which is selected
hideMenu	Boolean	Hides menu if set to true
deselectSiblings	Boolean	Deselects the siblings of the currently selected menu item
focusTrigger	Boolean	Sets focus to the ui:menuTrigger component

ui:menuTriggerPress

Indicates that a menu trigger is clicked.

For example, the `ui:menuTrigger` component registers this event so it can be fired by the component.

```
<aura:registerEvent name="menuTriggerPress"  type="ui:menuTriggerPress"
                    description="The event fired when the trigger is clicked." />
```

You can handle this event in a component that extends `ui:menuTrigger`, such as in a `ui:menuTriggerLink` component instance.

```
<ui:menu>
    <ui:menuTriggerLink aura:id="trigger" label="Contacts"
menuTriggerPress="{!c.triggered}"/>
        <ui:menuList class="actionMenu" aura:id="actionMenu">
            <ui:actionMenuItem aura:id="item1" label="All Contacts"
click="{!c.doSomething}"/>
            <ui:actionMenuItem aura:id="item2" label="All Primary" click="{!c.doSomething}"/>

        </ui:menuList>
</ui:menu>
```

This client-side controller retrieves the label of the trigger when it's clicked.

```
({
    triggered : function(component, event, helper) {
        var trigger = component.find("trigger");

        // Get the label on the trigger
        var triggerLabel = trigger.get("v.label");
    }
})
```

ui:validationError

Indicates that the component has validation errors.

To set a handler for the `ui:validationError` event, use the `onError` system attribute on a component that extends `ui:input`, such as `ui:inputNumber`.

The following `ui:inputNumber` component handles an error when the `ui:button` component is pressed. You can fire and handle these events in a client-side controller.

```
<aura:component>
    Enter a number:
    <!-- onError calls your client-side controller to handle a validation error -->
    <!-- onClearErrors calls your client-side controller to handle clearing of errors -->

    <ui:inputNumber aura:id="inputCmp" onError="{!c.handleError}"
onClearErrors="{!c.handleClearError}"/> <br/>

    <!-- press calls your client-side controller to trigger validation errors -->
    <ui:button label="Submit" press="{!c.doAction}"/>
</aura:component>
```

For more information, see Validating Fields on page 140.

Attribute Name	Type	Description
errors	Object[]	An array of error messages

System Event Reference

System events are fired by the framework during its lifecycle. You can handle these events in your Lightning apps or components, and within Salesforce1. For example, these events enable you to handle attribute value changes, URL changes, or when the app or component is waiting for a server response.

aura:doneRendering

Indicates that the initial rendering of the root application or root component has completed.

This event is automatically fired if no more components need to be rendered or rerendered due to any attribute value changes. The `aura:doneRendering` event is handled by a client-side controller. A component can have only one `<aura:handler event="doneRendering">` tag to handle this event.

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

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

```
<aura:component>
    <aura:handler event="aura:doneRendering" action="{!c.doneRendering}"/>
    <aura:attribute name="isDoneRendering" type="Boolean" default="false"/>
    <!-- Other component markup here -->
    <p>My component</p>
</aura:component>
```

This client-side controller checks that the `aura:doneRendering` event has been fired only once.

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

📝 Note: When `aura:doneRendering` is fired, `component.isRendered()` returns `true`. To check if your element is visible in the DOM, use utilities such as `component.getElement()`, `component.hasClass()`, or `element.style.display`.

The `aura:doneRendering` handler contains these required attributes.

Attribute Name	Type	Description
event	String	The name of the event, which must be set to `aura:doneRendering`.
action	Object	The client-side controller action that handles the event.

aura:doneWaiting

Indicates that the app or component is done waiting for a response to a server request. This event is preceded by an `aura:waiting` event. This event is fired after `aura:waiting`.

This event is automatically fired if no more response from the server is expected. The `aura:doneWaiting` event is handled by a client-side controller. A component can have only one `<aura:handler event="aura:doneWaiting">` tag to handle this event.

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

This example hides a spinner when `aura:doneWaiting` is fired.

```
<aura:component>
    <aura:handler event="aura:doneWaiting" action="{!c.hideSpinner}"/>
    <!-- Other component markup here -->
```

```
    <center><ui:spinner aura:id="spinner"/></center>
</aura:component>
```

This client-side controller fires an event that hides the spinner.

```
({
    hideSpinner : function (component, event, helper) {
        var spinner = component.find('spinner');
        var evt = spinner.get("e.toggle");
        evt.setParams({ isVisible : false });
        evt.fire();
    }
})
```

The `aura:doneWaiting` handler contains these required attributes.

Attribute Name	Type	Description
event	String	The name of the event, which must be set to `aura:doneWaiting`.
action	Object	The client-side controller action that handles the event.

aura:locationChange

Indicates that the hash part of the URL has changed.

This event is automatically fired when the hash part of the URL has changed, such as when a new location token is appended to the hash. The `aura:locationChange` event is handled by a client-side controller. A component can have only one `<aura:handler event="aura:locationChange">` tag to handle this event.

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

This client-side controller handles the `aura:locationChange` event.

```
({
    update : function (component, event, helper) {
        // Get the new location token from the event
        var loc = event.getParam("token");
        // Do something else
    }
})
```

The `aura:locationChange` handler contains these required attributes.

Attribute Name	Type	Description
event	String	The name of the event, which must be set to `aura:locationChange`.
action	Object	The client-side controller action that handles the event.

The `aura:locationChange` event contains these attributes.

Attribute Name	Type	Description
token	String	The hash part of the URL.
querystring	Object	The query string portion of the hash.

aura:systemError

Indicates that an error has occurred.

This event is automatically fired when an error is encountered during the run of a server-side action. The `aura:systemError` event is handled by a client-side controller. A component can have only one `<aura:handler event="aura:systemError">` tag to handle this event.

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

This example shows a button that triggers an error and a handler for the `aura:systemError` event.

```
<aura:component controller="namespace.myController">
    <aura:handler event="aura:systemError" action="{!c.showSystemError}"/>
    <aura:attribute name="response" type="Aura.Action"/>
    <!-- Other component markup here -->
    <ui:button aura:id="trigger" label="Trigger error" press="{!c.trigger}"/>
</aura:component>
```

This client-side controller triggers the firing of an error and handles that error.

```
({
    trigger: function(cmp, event) {
        // Call an Apex controller that throws an error
        var action = cmp.get("c.throwError");
        action.setCallback(cmp, function(response){
            cmp.set("v.response", response);
        });
        $A.enqueueAction(action);
    },

    showSystemError: function(cmp, event) {
        // Handle system error
        $A.log(cmp);
        $A.log(event);
    }
})
```

The `aura:systemError` handler contains these required attributes.

Attribute Name	Type	Description
event	String	The name of the event, which must be set to `aura:systemError`.
action	Object	The client-side controller action that handles the event.

The `aura:systemError` event contains these attributes. You can retrieve the attribute values using `event.getParam("message")`.

Attribute Name	Type	Description
message	String	The error message.
error	String	The error object.

aura:valueChange

Indicates that a value has changed.

This event is automatically fired when an attribute value changes. The `aura:valueChange` event is handled by a client-side controller. A component can have multiple `<aura:handler name="change">` tags to detect changes to different attributes.

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

This example updates a Boolean value, which automatically fires the `aura:valueChange` event.

```
<aura:component>
    <aura:attribute name="myBool" type="Boolean" default="true"/>

    <!-- Handles the aura:valueChange event -->
    <aura:handler name="change" value="{!v.myBool}" action="{!c.handleValueChange}"/>
    <ui:button label="change value" press="{!c.changeValue}"/>
</aura:component>
```

These client-side controller actions trigger the value change and handle it.

```
({
    changeValue : function (component, event, helper) {
      component.set("v.myBool", false);
    },

    handleValueChange : function (component, event, helper) {
        //handle value change
    }
})
```

The `change` handler contains these required attributes.

Attribute Name	Type	Description
name	String	The name of the handler, which must be set to `change`.
value	Object	The value for which you want to detect changes.
action	Object	The client-side controller action that handles the value change.

aura:valueDestroy

Indicates that a value is being destroyed.

This event is automatically fired when an attribute value is being destroyed. The `aura:valueDestroy` event is handled by a client-side controller. A component can have only one `<aura:handler name="destroy">` tag to handle this event.

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

This client-side controller handles the `aura:valueDestroy` event.

```
({
    valueDestroy : function (component, event, helper) {
      var val = event.getParam("value");
      // Do something else here
    }
})
```

For example, let's say that you are viewing your Lightning component in the Salesforce1 app. This `aura:valueDestroy` event is triggered when you tap on a different menu item on the Salesforce1 navigation menu, and your component is destroyed. In this example, the `token` attribute returns the component that's being destroyed.

The `destroy` handler contains these required attributes.

Attribute Name	Type	Description
name	String	The name of the handler, which must be set to `destroy`.
value	Object	The value for which you want to detect the event for.
action	Object	The client-side controller action that handles the value change.

The `aura:valueDestroy` event contains these attributes.

Attribute Name	Type	Description
value	String	The value being destroyed, which is retrieved via `event.getParam("value")`.

aura:valueInit

Indicates that a value has been initialized. This event is triggered on app or component initialization.

This event is automatically fired when an app or component is initialized, prior to rendering. The `aura:valueInit` event is handled by a client-side controller. A component can have only one `<aura:handler name="init">` tag to handle this event.

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

For an example, see Invoking Actions on Component Initialization on page 146.

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

The `init` handler contains these required attributes.

Attribute Name	Type	Description
name	String	The name of the handler, which must be set to `init`.
value	Object	The value that is initialized, which must be set to `{!this}`.
action	Object	The client-side controller action that handles the value change.

285

aura:waiting

Indicates that the app or component is waiting for a response to a server request. This event is fired before `aura:doneWaiting`.

This event is automatically fired when a server-side action is added using `$A.enqueueAction()` and subsequently run, or when it's expecting a response from an Apex controller. The `aura:waiting` event is handled by a client-side controller. A component can have only one `<aura:handler event="aura:waiting">` tag to handle this event.

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

This example shows a spinner when `aura:waiting` is fired.

```
<aura:component>
     <aura:handler event="aura:waiting" action="{!c.showSpinner}"/>
    <!-- Other component markup here -->
    <center><ui:spinner aura:id="spinner"/></center>
</aura:component>
```

This client-side controller fires an event that displays the spinner.

```
({
    showSpinner : function (component, event, helper) {
        var spinner = component.find('spinner');
        var evt = spinner.get("e.toggle");
        evt.setParams({ isVisible : true });
        evt.fire();
    }
})
```

The `aura:waiting` handler contains these required attributes.

Attribute Name	Type	Description
event	String	The name of the event, which must be set to `aura:waiting`.
action	Object	The client-side controller action that handles the event.

Supported HTML Tags

An HTML tag is treated as a first-class component by the framework. Each HTML tag is translated into 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

SEE ALSO:

Supporting Accessibility

Supported aura:attribute Types

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

Attribute Name	Type	Description
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 on page 146.
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`.
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(response) {
            var state = response.getState();
            if (state === "SUCCESS") {
                var stringItems = response.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" />
```

289

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();
```

> **Note:** Make your Apex class methods, getter and setter methods, available to your components by annotating them with `@AuraEnabled`.

SEE ALSO:

Working with Salesforce Records

Standard and Custom Object Types

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

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

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

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

> **Note:** Make your Apex class methods, getter and setter methods, available to your components by annotating them with `@AuraEnabled`.

SEE ALSO:

Working with Salesforce Records

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"`	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, `{ }`.

type	Example	Description
	default="{ a: 'label1', b: 'label2' }" />	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, use `aura:iteration`.

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" />
```

Using Arrays

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

This `aura:attribute` tag shows the syntax for an array of Apex objects:

```
<aura:attribute name="colorPalette" type="docSampleNamespace.Color[]" />
```

📝 Note: Make your Apex class methods, getter and setter methods, available to your components by annotating them with `@AuraEnabled`.

SEE ALSO:

[Working with Salesforce Records](#)

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.

type	Example	Description

`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