**SLDS – Salesforce Lightning Design System**

This is not part of the training, but see Resources for more info

It is a tool that allows UI Designers to design the ui so developers will focus more on the coding (but developers also handle the css)

**Lightning Experience** – a new modern UX and interface for using Salesforce

**Salesforce Classic** – considered to be the opposite of Lightning Experience

**Visualforce** – hosted on the Lightning platform

The Lightning Component framework is a UI framework for developing web apps for mobile and desktop devices. It’s a modern framework for building single-page applications with dynamic, responsive user interfaces for Lightning Platform apps. It uses JavaScript on the client side and Apex on the server side.

<https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/components_markup.htm>

<https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/components_bundle.htm>

<https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/components_html.htm>

Main goal: build a Salesforce platform for mobile

Lightning components must be published in a container.

Note: a Lightning Component app is NOT the same as a Salesforce app

UI Options:

Visualforce – Standard UI for Salesforce, but not ideal for mobile apps (requires a lot of custom code)

Lightning – Specifically optimized to perform well on mobile devices but it does NOT have the notion of an empty page. Also data access and security are pretty specific. You would have to duplicate a lot of code. Angular is platform agnostic, but lightning is Salesforce specific

Note: You can also create a lightning app which has lightning components on it, but an app can’t contain another app and a component can’t contain an app. Summary: a lightning component App is a container for lightning components

External frameworks (Angular, React, etc) – Ideal with a Visualforce container page. Very cumbersome with lightning components.

This part is confusing:

* You can’t add apps to Lightning Experience or the Salesforce app—you can only add components.
* What you add to App Launcher is a *Salesforce* app, which wraps up an Aura component, something defined in a <aura:component>. An Aura components app—that is, something defined in a <aura:application> —can’t be used to create Salesforce apps. A bit weird, but there it is.

Aura:

<https://trailhead.salesforce.com/lex_dev/lex_dev_overview/lex_dev_overview_future>

Lightning consists of :

Lightning Experience Platform

Lightning Web Component Framework

Lightning Data Service

User Interface API

<https://developer.salesforce.com/docs/atlas.en-us.224.0.uiapi.meta/uiapi/ui_api_get_started.htm>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.uiapi.meta/uiapi>

<https://trailhead.salesforce.com/trails/force_com_dev_intermediate/modules/api_basics>

<https://trailhead.salesforce.com/help?article=My-Domain-is-Already-Turned-On-in-Your-Trailhead-Playground&search=my%20domain>

<https://developer.salesforce.com/devcenter/lightning>

<https://trailhead.salesforce.com/lex_dev/lex_dev_overview/lex_dev_overview_future>

<https://www.youtube.com/watch?v=qm_kQFfRIaY>

* [Lightning Developer Center](https://developer.salesforce.com/devcenter/lightning)
* Trailhead: “User Interface Development Considerations” in the [Lightning Experience Development](https://trailhead.salesforce.com/lex_dev/lex_dev_overview/lex_dev_overview_future) module
* Video: [Lightning Intro Screencasts: Lightning Components](https://www.youtube.com/watch?v=qm_kQFfRIaY)
* [Use Aura Components in Lightning Experience and the Salesforce Mobile App](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/components_using_lex_s1.htm)
* [Configure Components for Lightning Pages and the Lightning App Builder](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/components_config_for_app_builder.htm)
* [Configure Components for Communities](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/components_config_for_builder.htm)
* [Use Lightning Components in Visualforce Pages](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/components_visualforce.htm)
* [Add Lightning Components to Any App with Lightning Out](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/lightning_out.htm)
* [Use Aura Components with Flows](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/components_using_flow.htm)
* When writing markup, you can add a component to an app, but you **can’t** add an app to another app, or an app to a component.
* An app has a standalone URL that you can access while testing, and which you can publish to your users. We often refer to these standalone apps as “my.app.”
* You can’t add apps to Lightning Experience or the Salesforce app—you can only add components. After the last unit this might sound weird; what exactly do you add to the App Launcher, if not an app? What you add to App Launcher is a *Salesforce* app, which wraps up an Aura component, something defined in a <aura:component>. An Aura components app—that is, something defined in a <aura:application> —can’t be used to create Salesforce apps. A bit weird, but there it is.

Developer Console

File > New Lightning Component

Creates a .cmp (component) file

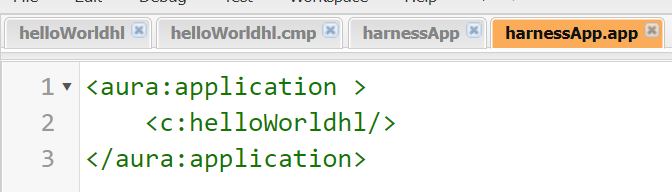
Can’t preview it. It needs to run in a container page



File > New Lightning Application

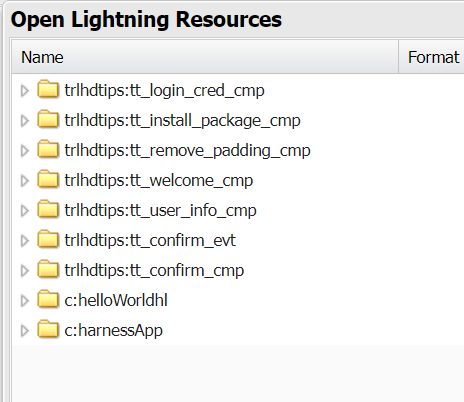
This will include the component helloWorldhl:

Note 2 tabs: harnessApp and harnessApp.app



To open existing:

File > Open Lightning Resources



Salesforce DX allows you to do UI development using a CLI or in VS using Salesforce Extensions

<https://developer.salesforce.com/platform/dx>

sfdx -

My Domain is required to create custom Lightning components and set up single sign-on (SSO) in an org. It’s so important that all production, Developer Edition, and trial orgs created in Winter ’21 org later get a My Domain by default. To learn how to activate My Domain in your production org, see the [User Authentication](https://trailhead.salesforce.com/modules/identity_login) module. To learn more about My Domain and your Trailhead Playground, check out this [knowledge article](https://trailhead.salesforce.com/help?article=My-Domain-is-Already-Turned-On-in-Your-Trailhead-Playground&search=my%20domain).

Links for the Sample React app that calls the UI API

<https://github.com/forcedotcom/RecordViewer>

<https://reactjs.org/>

<https://www.valentinog.com/blog/react-redux-tutorial-beginners/>

<https://redux-saga.js.org/docs/introduction/BeginnerTutorial.html>

<https://www.sohamkamani.com/blog/2017/03/31/react-redux-connect-explained/>

Issues with the Sample Instructions

<https://help.salesforce.com/s/articleView?id=sf.connected_app_create_api_integration.htm&type=5>

lightning components should call LDS (lightning data service) instead of UI API (LDS is built on top of UI API)

Setup > Object Manager >

Select Object > Page Layouts

<https://developer.salesforce.com/docs/atlas.en-us.224.0.uiapi.meta/uiapi>

https://{your\_instance}.salesforce.com/services/data/v{api\_version}/ui-api

**Connected Apps:** To call Salesforce APIs from external apps

Comments because the instructions sucked:

I also battled with this module.   
  
1. Docker is NOT compatible with Windows 10 Home. Rather use Docker Toolbox if you are on Windows 10 Home.  
  
2. Make sure you add your RecordViewer folder to the Shared Folders using VirtualBox / Settings.  
  
3. The compile command works fine as long as your DEFAULT box is running ("docker-machine start" at command prompt).  
  
4. Instead of using LOCALHOST:8443, replace all references with the IP Number of the DEFAULT virtual box. You can check the IP with command "docker-machine ls".  
  
For me, it was https://192.168.92.100:8443/  
  
Use this URL when setting up the connected app in the APP MANAGER and in the CORS setup.  
  
Lastly, if you are connecting to a PLAYGROUND, enter the URL of your playground when authorising, and NOT the generic login.salesforce.com as shown in the screenshot.

----------------------------------------

In your Trailhead Playground, from **Setup**, enter App and select **App Manager**.  
Click **New Connected App** and enter these settings:  
Connected App Name: **RecordViewer**  
Contact Email: **Your email address**  
Select: **Enable OAuth Settings and enter these settings**:  
Callback **URL: https://localhost:8443/oauth-redirect**  
Selected **OAuth Scopes: Access and manage your data (api)**  
Click **Save**.  
On the New Connected App page, click **Continue**.  
  
----------------  
  
  
From **Setup**, enter **CORS**and select **CORS**.  
Click **New**.  
For Origin URL Pattern, enter **https://localhost:8443**  
Click **Save**.  
  
  
-------------------   
and finally to finish the challenge:  
  
Go to Accounts and click **New**  
Account name: **ABC Computing**  
Rating: **Warm**  
**Save.**

=========================================

Constructing the UI API URL:

let recordViewUrl = action.creds.instanceUrl + '/services/data/v48.0/ui-api/record-ui/'

+ action.recordId + '?formFactor=' + action.context.formFactor + '&layoutTypes=Full&modes=View,Edit';

The **form factor** changes the layout of the fields. Choose a form factor that matches the type of hardware the app is running on. Large is a layout for desktop clients. Medium is a layout for tablets, and Small is a layout for phones. In the large and medium form factors, sections have a two-column layout. In the small form factor, sections have a one-column layout.

The **layout type** determines how many fields are returned. The possible layout types are Full and Compact. The default full layout includes all the fields from the page layout assigned to the current user. The compact layout includes all the fields from the compact layout assigned to the current user. You can edit both layout types in Setup. Regardless of layout type, the response includes only fields that the user has access to.

The **mode** corresponds to the task the user is performing: Create, View, or Edit. The layout information is different depending on the mode. For example, in create mode, the layout doesn’t include the System Information section, which includes fields like Created By and Last Modified By.

Return object with children:

/ui-api/record-ui/001R0000003IG0vIAG?childRelationships=Account.Opportunities

**Workbench – Salesforce version of Postman or Swagger?**

<https://workbench.developerforce.com/>

Workbench is a suite of tools for interacting with your Salesforce org through the API. Because you can make REST requests from any HTTP sender, there are plenty of other tools available for you to use (for example, check out cURL or Postman). But because Workbench provides a friendly framework specifically for Salesforce APIs, it’s the perfect way to dig in before you’re ready to write a full-on integration.

The first step is to log in to Workbench.

1. Log in to your Trailhead Playground. In another browser tab, navigate to [Workbench](https://workbench.developerforce.com/).
2. For Environment, select **Production**.
3. For API Version, select the highest available number.
4. Make sure that you select **I agree to the terms of service**.
5. Click **Login with Salesforce**.

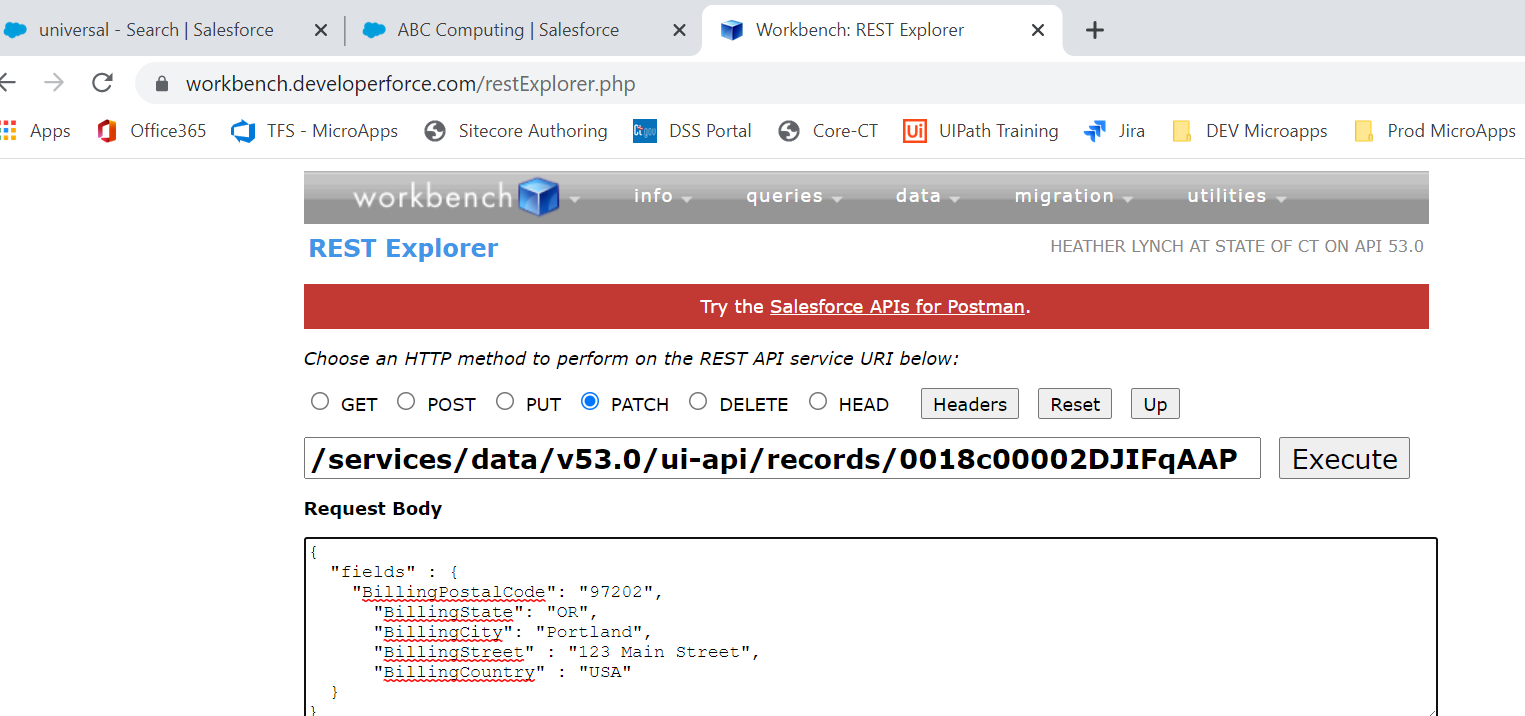
Utilities > REST Explorer

**ETags**

ETag is an HTTP header that uniquely identifies a response. Your app can pass an ETag back to Salesforce using the If-None-Match HTTP request header.

If the JSON that Salesforce is going to send back is the same as what you have, you get an HTTP 304, which means that nothing has changed, and your app doesn’t have to deserialize another payload.

Update the billing address on this (specific) Account (got the id for the account record by opening the account in Salesforce)

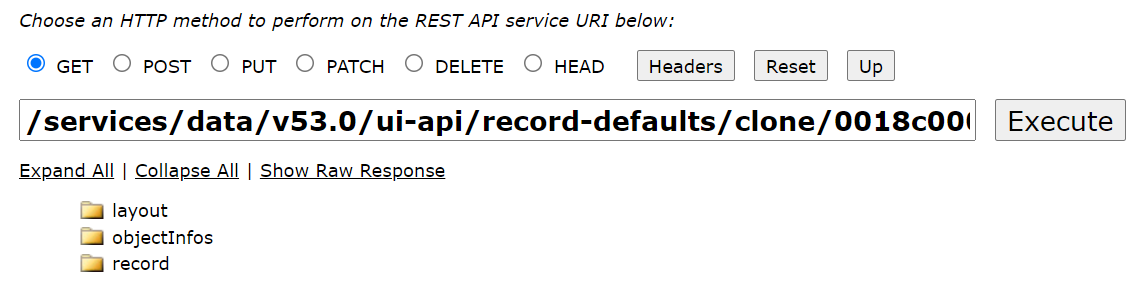


Get information from an existing record so you can clone it

GET /ui-api/record-defaults/clone/{recordId}

Return includes meta dta about the layout and the data values

* Includes **layout** metadata for the object in edit mode.
* Includes **objectInfos** (object metadata) for the cloned object and for any nested objects. For example, if you clone an Opportunity, the response includes object metadata for the Opportunity object. It also includes object metadata for the Account object, because the opportunity references an account. It also includes object metadata for the User object, because fields like OwnerId reference a user.
* Includes the default **record** data from the record you’re cloning so you can use it to create a record.



create or clone:

POST /ui-api/records

{

"apiName": "Account",

"fields": {

"Name": "New Universal Containers"

}

}

Get Defaults to create or clone

GET /ui-api/record-defaults/create/{objectApiName}

* objectApiName—The API name of the object type for the new record.

/ui-api/record-defaults/clone/{recordId}

<https://developer.salesforce.com/docs/atlas.en-us.224.0.uiapi.meta/uiapi/ui_api_resources_record_defaults_create.htm>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.uiapi.meta/uiapi/ui_api_resources_record_defaults_clone.htm>

Dependent Picklists (child list changes based on parent selection)



GET /ui-api/object-info/{objectApiName}/picklist-values/{recordTypeId}

<https://developer.salesforce.com/docs/atlas.en-us.224.0.uiapi.meta/uiapi/ui_api_features_records_dependent_picklist.htm>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.uiapi.meta/uiapi/ui_api_features_records_dependent_picklist.htm>

Attributes:

Attributes are like instance variables

<aura:component>

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

<p>{!'Hello! ' + v.message}</p>

</aura:component>

v is something called a value provider. Value providers are a way to group, encapsulate, and access related data. Value providers are a complicated topic, so for now, think of v as an automatic variable that’s made available for you to use. In our component, v is a value provider for the view, which is the helloMessage component itself.

Using Objects:

<aura:component>

<aura:attribute name="expense" type="Expense\_\_c"/>

<p>Amount:

<lightning:formattedNumber value="{!v.expense.Amount\_\_c}" style="currency"/>

</p>

<p>

Client: {!v.expense.Client\_\_c}

</p>

<p>

<lightning:input type="toggle"

label="Reimbursed?"

name="reimbursed"

checked="{!v.expense.Reimbursed\_\_c}" />

</p>

<!-- Other markup here -->

</aura:component>

----------------------------------------------------------------------------------------------------------

Another example:

<aura:component>

<aura:attribute name="messages" type="List"

default="['You look nice today.',

'Great weather we\'re having.',

'How are you?']"/>

<h1>Hello Playground</h1>

<p>Silly fun with attributes and expressions.</p>

<h2>List Items</h2>

<p><c:helloMessage message="{!v.messages[0]}"/></p>

<p><c:helloMessage message="{!v.messages[1]}"/></p>

<p><c:helloMessage message="{!v.messages[2]}"/></p>

<h2>List Iteration</h2>

<aura:iteration items="{!v.messages}" var="msg">

<p><c:helloMessage message="{!msg}"/></p>

</aura:iteration>

<h2>Conditional Expressions and Global Value Providers</h2>

<aura:if isTrue="{!$Browser.isIPhone}">

<p><c:helloMessage message="{!v.messages[0]}"/></p>

<aura:set attribute="else">

<p><c:helloMessage message="{!v.messages[1]}"/></p>

</aura:set>

</aura:if>

</aura:component>

* [Component Attributes](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/components_attributes.htm)
* [Supported aura:attribute Types](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/ref_aura_attribute.htm)
* [Basic Types](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/ref_attr_types_basic.htm)
* [Standard and Custom Object Types](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/ref_attr_types_object_salesforce.htm)
* [Using Expressions](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/expr_overview.htm)
* [Value Providers](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/expr_source.htm)
* [Expression Evaluation](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/expr_eval.htm)
* [Expression Operators Reference](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/expr_operators.htm)
* [Expression Functions Reference](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/expr_functions.htm)

Controllers:

<aura:component>

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

<p>Message of the day: {!v.message}</p>

<div>

<lightning:button label="You look nice today."

onclick="{!c.handleClick}"/>

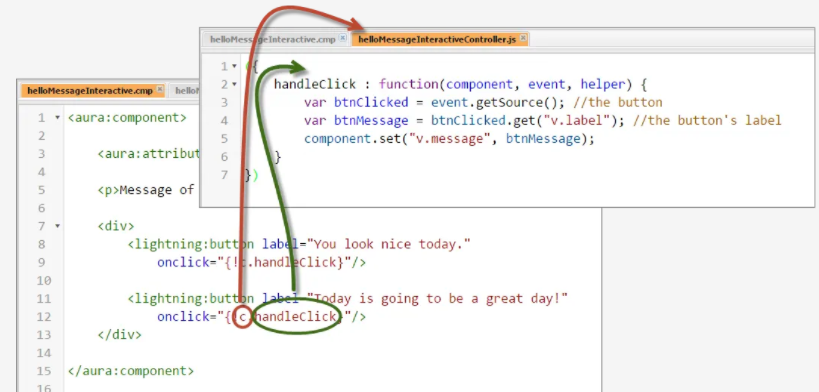
<lightning:button label="Today is going to be a great day!"

onclick="{!c.handleClick}"/>

</div>

</aura:component>

handleClick method is implemented in the component controller



When the button is clicked, its action handler gets called (1). In the action handler, the controller gets the button that was clicked, pulls the label text out of it, and then sets the component’s message attribute to that text (2). And the message of the day is updated (3). Today is going to be a great day!

While it’s not technically required, you should *always* declare your controller functions to take these three parameters. We’ll talk more about them as we go, but for now, these parameters represent:

* component—the component. In this case, it’s helloMessageInteractive.
* event—the event that caused the action handler to be called.
* helper—the component’s helper, another JavaScript resource of reusable functions.

Remember that handleClick is connected to our <lightning:button> tag and its onclick attribute. The event, then, is someone clicking the button. Inside that event it has the notion of a source, the thing that generated the event, which is the button itself. So, calling event.getSource() gets us a reference to the specific <lightning:button> that was clicked.

You can call get() on any component and provide the name of the attribute you want to retrieve, in the format v.*attributeName*. The result is the attribute value.

Multiple action handlers require a comma in between

handleClick2: function(component, event, helper) {

let newMessage = event.getSource().get("v.label");

component.set("v.message", newMessage);

},

handleClick3: function(component, event, helper) {

component.set("v.message", event.getSource().get("v.label"));

}

Logging:

console.**log**(**"handleClick2: Message: "** + newMessage);

goes to the browser’s javascript console

* [Working with Base Lightning Components](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/lightning_overview.htm)
* [Event Handling in Base Lightning Components](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/lightning_base_events.htm)
* [lightning:button Documentation](https://developer.salesforce.com/docs/component-library/bundle/lightning:button/documentation)
* [lightning:input Documentation](https://developer.salesforce.com/docs/component-library/bundle/lightning:input/documentation)
* [Actions and Events](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/events_and_actions.htm)
* [Handling Events with Client-Side Controllers](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/js_client_side_controller.htm)
* [Which Button Was Pressed?](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/js_cb_which_button_pressed.htm)
* [Enable Debug Mode for Lightning Components](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/aura_debug_mode.htm)

Example:

Button will change the value of this toggle:

<aura:component >

<aura:attribute name="item" type="Camping\_Item\_\_c" required="true"/>

<p>

Name: {!v.item.Name}

</p>

<p>Price:

<lightning:formattedNumber value="{!v.item.Price\_\_c}" style="currency"/>

</p>

<p>Quantity:

<lightning:formattedNumber value="{!v.item.Quantity\_\_c}"/>

</p>

<p>

<lightning:input type="toggle"

label="Packed?"

name="packed"

checked="{!v.item.Packed\_\_c}" />

</p>

<lightning:button label="Packed!"

onclick="{!c.packItem}"/>

</aura:component>

-------------------------------------------------------------------------------------

Controller:

({

packItem : function(component, event, helper) {

component.set("v.item.Packed\_\_c", true);

component.set("v.packed", event.getSource().set("v.disabled",true));

}

})

========================================================

Create an app from the ground up:

1. File > new > Lightning Application

expenseApp:

<aura:application extends="force:slds">

<!-- This component is the real "app" -->

<c:expenses/>

</aura:application>

1. File > New > Lightning Component

expenses (included in the above app)

NOTE: This would actually include smaller components, but is one big component here:

<aura:component>

<!-- PAGE HEADER -->

<lightning:layout class="slds-page-header slds-page-header\_object-home">

<lightning:layoutItem>

<lightning:icon iconName="standard:scan\_card" alternativeText="My Expenses"/>

</lightning:layoutItem>

<lightning:layoutItem padding="horizontal-small">

<div class="page-section page-header">

<h1 class="slds-text-heading\_label">Expenses</h1>

<h2 class="slds-text-heading\_medium">My Expenses</h2>

</div>

</lightning:layoutItem>

</lightning:layout>

<!-- / PAGE HEADER -->

<!-- NEW EXPENSE FORM -->

<lightning:layout>

<lightning:layoutItem padding="around-small" size="6">

<!-- CREATE NEW EXPENSE -->

<div aria-labelledby="newexpenseform">

<!-- BOXED AREA -->

<fieldset class="slds-box slds-theme\_default slds-container\_small">

<legend id="newexpenseform" class="slds-text-heading\_small

slds-p-vertical\_medium">

Add Expense

</legend>

<!-- CREATE NEW EXPENSE FORM -->

<form class="slds-form\_stacked">

<lightning:input aura:id="expenseform" label="Expense Name"

name="expensename"

value="{!v.newExpense.Name}"

required="true"/>

<lightning:input type="number" aura:id="expenseform" label="Amount"

name="expenseamount"

min="0.1"

formatter="currency"

step="0.01"

value="{!v.newExpense.Amount\_\_c}"

messageWhenRangeUnderflow="Enter an amount that's at least $0.10."/>

<lightning:input aura:id="expenseform" label="Client"

name="expenseclient"

value="{!v.newExpense.Client\_\_c}"

placeholder="ABC Co."/>

<lightning:input type="date" aura:id="expenseform" label="Expense Date"

name="expensedate"

value="{!v.newExpense.Date\_\_c}"/>

<lightning:input type="checkbox" aura:id="expenseform" label="Reimbursed?"

name="expreimbursed"

checked="{!v.newExpense.Reimbursed\_\_c}"/>

<lightning:button label="Create Expense"

class="slds-m-top\_medium"

variant="brand"

onclick="{!c.clickCreate}"/>

</form>

<!-- / CREATE NEW EXPENSE FORM -->

</fieldset>

<!-- / BOXED AREA -->

</div>

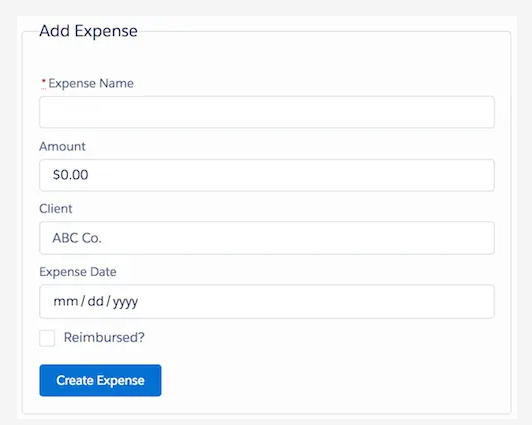
<!-- / CREATE NEW EXPENSE --> </lightning:layoutItem>

</lightning:layout>

<!-- / NEW EXPENSE FORM -->

</aura:component>

1. Result looks like this (note the controller has not been coded yet):



Sample attribute object:

<aura:attribute name="newExpense" type="Expense\_\_c"

default="{ 'sobjectType': 'Expense\_\_c',

'Name': '',

'Amount\_\_c': 0,

'Client\_\_c': '',

'Date\_\_c': '',

'Reimbursed\_\_c': false }"/>

Action Handler:

({

clickCreate: function(component, event, helper) {

let validExpense = component.find('expenseform').reduce(function (validSoFar, inputCmp) {

// Displays error messages for invalid fields

inputCmp.showHelpMessageIfInvalid();

return validSoFar && inputCmp.get('v.validity').valid;

}, true);

// If we pass error checking, do some real work

if(validExpense){

// Create the new expense

let newExpense = component.get("v.newExpense");

console.log("Create expense: " + JSON.stringify(newExpense));

helper.createExpense(component, newExpense);

}

}

})

component.find() only lets you access the component and its child components from the controller and helper. It isn’t a magic way to go wandering around the component hierarchy and read or change things. Remember, components are supposed to be self-contained

Helpers is an advanced concept not covered here:

* A component’s helper is the appropriate place to put code to be shared between several different action handlers.
* A component’s helper is a great place to put complex processing details, so that the logic of your action handlers remains clear and streamlined.
* Helper functions can have any function signature. That is, they’re not constrained the way that action handlers in the controller are. (Why is this? Because *you* are calling the helper function directly from your code. By contrast, the framework calls action handlers via the framework runtime.) It’s a convention and recommended practice to always provide the component as the first parameter to helper functions.

OK, let’s get on with it. In the Developer Console, click the **HELPER** button for the expenses component to create the associated helper resource, and then replace the example code with the following.

({

createExpense: function(component, expense) {

let theExpenses = component.get("v.expenses");

// Copy the expense to a new object

// THIS IS A DISGUSTING, TEMPORARY HACK

let newExpense = JSON.parse(JSON.stringify(expense));

theExpenses.push(newExpense);

component.set("v.expenses", theExpenses);

}

})

console.log("Expenses before 'create': " + JSON.stringify(theExpenses));

theExpenses.push(newExpense);

component.set("v.expenses", theExpenses);

console.log("Expenses after 'create': " + JSON.stringify(theExpenses));

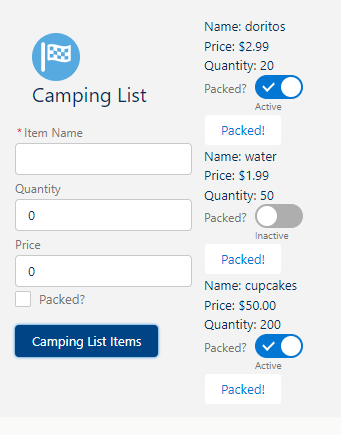
What component.set() does here isn’t update the value of the expenses attribute. It triggers notification that the expenses attribute has changed.

## Resources

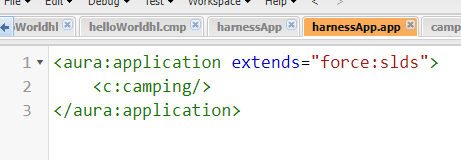
* [Working with Base Lightning Components](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/lightning_overview.htm)
* [Event Handling in Base Lightning Components](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/lightning_base_events.htm)
* [Actions and Events](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/events_and_actions.htm)
* [Handling Events with Client-Side Controllers](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/js_client_side_controller.htm)
* [Finding Components by ID](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/js_cb_find_by_id.htm)
* [Validating Fields](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/js_validate_fields.htm)
* [Lightning Design System Considerations](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/lightning_base_slds.htm)

Top of Form

Here is the code for this example:



1. Harness app:



1. camping component:

<aura:component >

<lightning:layout class="slds-page-header slds-page-header\_object-home">

<lightning:layoutItem>

<c:campingList/>

</lightning:layoutItem>

</lightning:layout>

</aura:component>

1. campingHeader

<aura:component >

<lightning:layout class="slds-page-header slds-page-header\_object-home">

<div class="page-section page-header">

<lightning:icon iconName="action:goal" alternativeText="Camping List"/>

<h1 class="slds-text-heading\_medium">Camping List</h1>

</div>

</lightning:layout>

</aura:component>

1. campingListItem

<aura:component >

<aura:attribute name="item" type="Camping\_Item\_\_c" required="true"/>

<p>

Name: {!v.item.Name}

</p>

<p>Price:

<lightning:formattedNumber value="{!v.item.Price\_\_c}" style="currency"/>

</p>

<p>Quantity:

<lightning:formattedNumber value="{!v.item.Quantity\_\_c}"/>

</p>

<p>

<lightning:input type="toggle"

label="Packed?"

name="packed"

checked="{!v.item.Packed\_\_c}" />

</p>

<lightning:button label="Packed!"

onclick="{!c.packItem}"/>

</aura:component>

1. campingList

<aura:component>

<aura:attribute name="items" type="Camping\_Item\_\_c[]" />

<aura:attribute name="newItem" type="Camping\_Item\_\_c" default="{

'sObjectType': 'Camping\_Item\_\_c',

'Quantity\_\_c': 0,

'Price\_\_c': 0,

'Packed\_\_c': false}" />

<lightning:layout >

<div style="display: flex; flex-direction: column;">

<c:campingHeader/>

<form class="slds-form--stacked">

<lightning:input aura:id="campingListForm" label="Item Name"

name="itemName"

type="text"

value="{!v.newItem.Name}"

required="true"/>

<lightning:input type="number" aura:id="campingListForm" label="Quantity"

name="Quantity"

min="1"

value="{!v.newItem.Quantity\_\_c}" />

<lightning:input aura:id="campingListForm" label="Price"

name="price"

formatter="currency"

value="{!v.newItem.Price\_\_c}" />

<lightning:input type="checkbox" aura:id="campingListForm" label="Packed?"

name="Packed"

checked="{!v.newItem.Packed\_\_c}"/>

<lightning:button label="Camping List Items" class="slds-m-top--medium" variant="brand" onclick="{!c.clickCreateItem}" />

</form>

</div>

<p class="slds-p-horizontal--small">

<aura:iteration items="{!v.items}" var="item">

<c:campingListItem item="{!item}" />

</aura:iteration>

</p>

</lightning:layout>

</aura:component>

1. campingListController:

({

clickCreateItem : function(component, event, helper) {

var validItem = component.find('campingListForm').reduce((validSoFar, inputCmp)=> {

inputCmp.showHelpMessageIfInvalid();

return validSoFar && inputCmp.get('v.validity').valid;

}, true);

if (validItem) {

var newItem = component.get('v.newItem');

console.log('New Item Created: ' +JSON.stringify(newItem));

var campingItems = component.get('v.items');

campingItems.push(JSON.parse(JSON.stringify(newItem)));

component.set('v.items', campingItems);

newItem = {'sObjectType': 'Camping\_Item\_\_c', 'Quantity\_\_c': 0, 'Price\_\_c': 0, 'Packed\_\_c': false};

component.set('v.newItem', newItem);

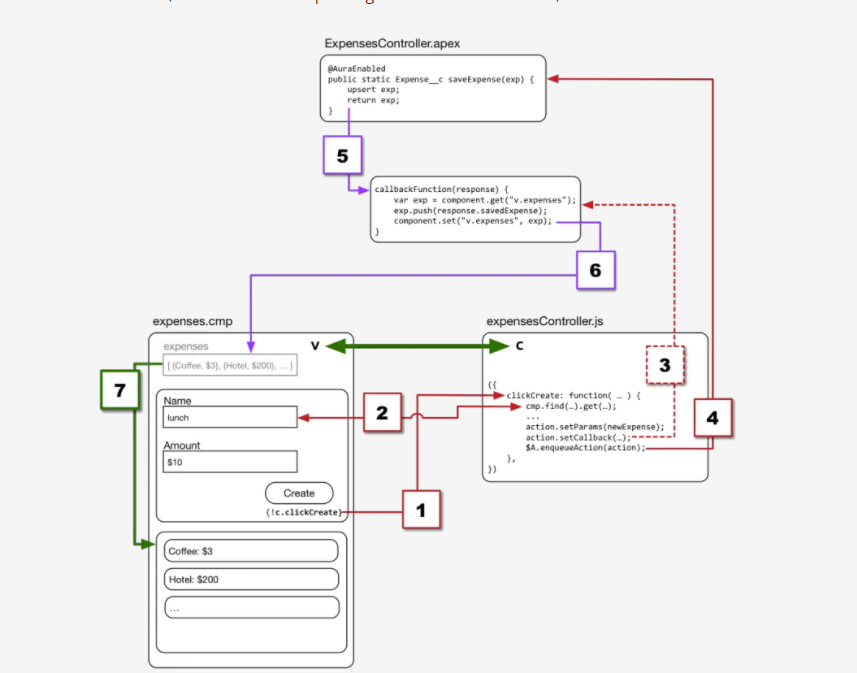
}

}

})

Everything above was client side

Server Side Processing:



Sample code to get data from Salesforce server:

// Load expenses from Salesforce

doInit: function(component, event, helper) {

// Create the action

let action = component.get("c.getExpenses");

// Add callback behavior for when response is received

action.setCallback(this, function(response) {

let state = response.getState();

if (state === "SUCCESS") {

component.set("v.expenses", response.getReturnValue());

}

else {

console.log("Failed with state: " + state);

}

});

// Send action off to be executed

$A.enqueueAction(action);

},

1. Create a remote method call.

let action = component.get("c.getExpenses");

\*\*\* c represents the remote Apex controller

Confusing: it could be client or server side

1. Set up what should happen when the remote method call returns.

action.setCallback…

It does not actually run yet…it is just a function definition

1. Queue up the remote method call.

$A.enqueueAction(action) adds the server call that we’ve just configured to the Aura component framework request queue. It, along with other pending server requests, will be sent to the server in the next request cycle.

* It queues up the server request.
* As far as your controller action is concerned, that’s the end of it.
* You’re not guaranteed when, or if, you’ll hear back.

This is standard javascript framework Async processing.

The callback function handles a response

function(response) {

let state = response.getState();

if (state === "SUCCESS") {

component.set("v.expenses", response.getReturnValue());

}

}

1. Get the state of the response.
2. If the state is SUCCESS, that is, our request completed as planned, then:
3. Set the component’s expenses attribute to the value of the response data.

* [Invoking Actions on Component Initialization](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/js_cb_init_handler.htm)
* [Creating Server-Side Logic with Controllers](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/controllers_server_intro.htm)
* [Calling a Server-Side Action](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/controllers_server_actions_call.htmer_actions_call.htm)
* [Working with Salesforce Records](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/apex_records.htm)
* [Securing Data in Apex Controllers](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/apex_crud_fls.htm)
* [Returning Errors from an Apex Server-Side Controller](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/controllers_server_apex_custom_errors.htm)
* [Queuing of Server-Side Actions](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/controllers_server_actions_queue.htm)
* [Testing Your Apex Code](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/apps_packaging_tests.htm)

public with sharing class ExpensesController {

// STERN LECTURE ABOUT WHAT'S MISSING HERE COMING SOON

@AuraEnabled

public static List<Expense\_\_c> getExpenses() {

return [SELECT Id, Name, Amount\_\_c, Client\_\_c, Date\_\_c,

Reimbursed\_\_c, CreatedDate

FROM Expense\_\_c];

}

}

* The @AuraEnabled annotation before the method declaration. “Aura” is the name of the framework at the core of Lightning Components. You’ve seen it used in the namespace for some of the core tags, such as <aura:component>. Now you know where it comes from.
* The static keyword. All @AuraEnabled controller methods must be static methods, and either public or global scope.

<aura:component controller="ExpensesController">

**Example Apex Controller:**

public with sharing class ExpensesController {

@AuraEnabled

public static List<Expense\_\_c> getExpenses() {

// Perform isAccessible() checking first, then

return [SELECT Id, Name, Amount\_\_c, Client\_\_c, Date\_\_c,

Reimbursed\_\_c, CreatedDate

FROM Expense\_\_c];

}

@AuraEnabled

public static Expense\_\_c saveExpense(Expense\_\_c expense) {

// Perform isUpdateable() checking first, then

upsert expense;

return expense;

}

}

Using the with sharing keywords is one of the essential security measures you need to take when writing server-side controller code. However, it’s a measure that’s necessary, but not sufficient. Do you see the comments about performing isAccessible() and isUpdateable() checks? with sharing only takes you so far. In particular, you need to implement object- and field-level security (which you’ll frequently see abbreviated to FLS) yourself.

For example, here’s a version of our getExpenses() method with this security minimally implemented.

@AuraEnabled

public static List<Expense\_\_c> getExpenses() {

// Check to make sure all fields are accessible to this user

String[] fieldsToCheck = new String[] {

'Id', 'Name', 'Amount\_\_c', 'Client\_\_c', 'Date\_\_c',

'Reimbursed\_\_c', 'CreatedDate'

};

Map<String,Schema.SObjectField> fieldDescribeTokens =

Schema.SObjectType.Expense\_\_c.fields.getMap();

for(String field : fieldsToCheck) {

if( ! fieldDescribeTokens.get(field).getDescribe().isAccessible()) {

throw new System.NoAccessException();

}

}

// OK, they're cool, let 'em through

return [SELECT Id, Name, Amount\_\_c, Client\_\_c, Date\_\_c,

Reimbursed\_\_c, CreatedDate

FROM Expense\_\_c];

}

It’s quite an expansion from our initial one-liner, and it’s still just adequate. Also, describe calls are expensive. If your app is calling this method frequently, you should find a way to optimize or cache your access checks per user.

Like with SLDS, we simply don’t have the space to teach all of the details of secure Apex coding. Unlike with SLDS, taking responsibility for the security of the code you write is not optional. If you haven’t read the secure coding practices articles in the Resources, please add them to your queue.

OK, </stern-lecture>.

## Saving Data to Salesforce

Before we implement the Add Expense form for real, no cheating, let’s first look at how creating a new record is a different challenge from reading existing records. With doInit(), we simply read some data, and then updated the user interface of the app. Straightforward, even if we did have to get Carly Rae involved in the explanation.

Creating a new record is more involved. We’re going to read values from the form, create a new expense record locally, send that record off to be saved on the server, and then, when the server tells us it’s saved, update the user interface, using the record returned from the server.

Does that make it sounds like it’s going to be really complicated? Like, maybe we’re going to need the Rolling Stones and a whole album of songs to help us out with the next explanation?

Let’s take a look at some code, and you can decide for yourself.

First, make sure you’ve saved the updated version of the Apex controller, the preceding version that includes the saveExpense() method.

Remember when we showed you how to handle the form submission? When at least one of the fields is invalid, you’ll see an error message and the form isn’t submitted. The error message clears when all fields are valid.

Because we put all of the details of (and all the cheating on) creating a new expense into the helper createExpense() function, we don’t need to make any other changes in the controller. So far, so easy?

So, all we need to do is change the createExpense() function in the helper, to do all those complicated things we mentioned previously. Here’s that code.

createExpense: function(component, expense) {

let action = component.get("c.saveExpense");

action.setParams({

"expense": expense

});

action.setCallback(this, function(response){

let state = response.getState();

if (state === "SUCCESS") {

let expenses = component.get("v.expenses");

expenses.push(response.getReturnValue());

component.set("v.expenses", expenses);

}

});

$A.enqueueAction(action);

},

Is that as complicated as you were expecting? As many lines? We hope not!

In truth, there is only one new thing in this action handler, and it’s easy to understand. Let’s walk through the code.

We begin by creating the action, with component.get("c.saveExpense") getting the new Apex controller method. Very familiar.

Next we attach a data payload to the action. This is new. We need to send the data for the new expense up to the server. But look how easy it is! You just use action.setParams() and provide a JSON-style object with parameter name-parameter value pairs. The one trick, and it’s important, is that your parameter name must match the parameter name used in your Apex method declaration.

Next we set the callback for the request. Again, this is what will happen when the server returns a response. If you compare this callback function with our original createExpense helper function, it’s virtually identical (minus the disgusting hack).

Just as in the prior version, we get() the expenses attribute, push() a value onto it, and then set() it. The only real difference is, instead of push()ing our local version of the new expense into the array, we’re push()ing the server’s response!

Why does this work? Because the server-side method upserts the (in this case new) record, which stamps an ID on it, and then returns the resulting record. Once again the server-side and client-side data types match, so we don’t have to do any extra work.

And, well, that’s it. No Rolling Stones needed!

## Things to Watch Out For

While we’ve covered all of the essentials for connecting your client-side Aura component code with server-side Apex code, there are a couple of things that are kind of worth pointing out before they bite you in the you-know-where.

The first issue is case sensitivity, and this boils down to Apex and Salesforce in general are case-**in**sensitive, but JavaScript is case-**sensitive**. That is, “Name” and “name” are the same in Apex, but different in JavaScript.

This can and will lead to absolutely maddening bugs that are completely invisible to your eyes, even when they’re right in front of your face. Especially if you’ve been working with non-Lightning Components code on Salesforce for a while, you might no longer think about the case of object and field names, methods, and so on, at all.

So here’s a best practice for you: Always use the **exact** API name of every object, field, type, class, method, entity, element, elephant, or what have you. Always, everywhere, even when it doesn’t matter. That way, you won’t have problems. Or, at least, not this problem.

The other issue we’d like to draw your attention to is the nature of “required.” We can’t resist repeating a famous quotation: “You keep using that word. I do not think it means what you think it means.”

In the code we’ve written so far we’ve seen at least two different kinds of “required.” In the markup for the Add Expense form, you see the word used two ways. For example, on the expense name field.

<lightning:input aura:id="expenseform"

label="Expense Name"

name="expensename"

value="{!v.newExpense.Name}"

required="true"/>

The <lightning:input> tag has its required attribute set to true. These both illustrate only one meaning of required, which is “set the user interface of this element to indicate the field is required.” In other words, this is cosmetic only. There’s no protection for the quality of your data here.

Another meaning of the word “required” is illustrated in the validation logic we wrote for the same field.

let validExpense = component.find('expenseform').reduce(function (validSoFar, inputCmp) {

// Displays error messages for invalid fields

inputCmp.showHelpMessageIfInvalid();

return validSoFar && inputCmp.get('v.validity').valid;

}, true);

The word “required” is nowhere to be seen, but that’s what the validation logic enforces. You must set a value for the expense name field.

And, as far as it goes, this is great. Your expense form won’t submit a new expense with an empty name. Unless, you know, there’s a bug. Or, maybe some other widget uses your same server-side controller, but doesn’t do its form validation so carefully. And so on. So, this is some protection for your data quality, but it’s not perfect.

How do you enforce, and we mean enforce, a data integrity rule about, in this example, expense name? You do it server-side. And not just anywhere server-side. You put the rule in the field definition, or you encode it into a trigger. Or, if you’re a belt-and-suspenders kind of engineer, as all right thinking engineers are, both.

For true data integrity, when “required” means required, enforce it at the lowest level possible.

## Resources

* [Invoking Actions on Component Initialization](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/js_cb_init_handler.htm)
* [Creating Server-Side Logic with Controllers](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/controllers_server_intro.htm)
* [Calling a Server-Side Action](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/controllers_server_actions_call.htmer_actions_call.htm)
* [Working with Salesforce Records](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/apex_records.htm)
* [Securing Data in Apex Controllers](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/apex_crud_fls.htm)
* [Returning Errors from an Apex Server-Side Controller](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/controllers_server_apex_custom_errors.htm)
* [Queuing of Server-Side Actions](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/controllers_server_actions_queue.htm)
* [Testing Your Apex Code](https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/apps_packaging_tests.htm)