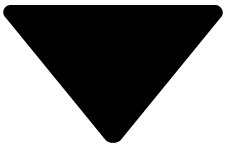
1. Aura Components Basics



2. Connect to Salesforce with Server-Side Controllers

Connect to Salesforce with Server-Side Controllers

Learning Objectives

After completing this unit, you'll be able to:

- Create Apex methods that can be called remotely from Aura component code.
- Make calls from Aura components to remote methods.
- Handle server responses asynchronously using callback functions.
- Stretch goal: explain the difference between "c.", "c:", and "c.".

Server-Side Controller Concepts

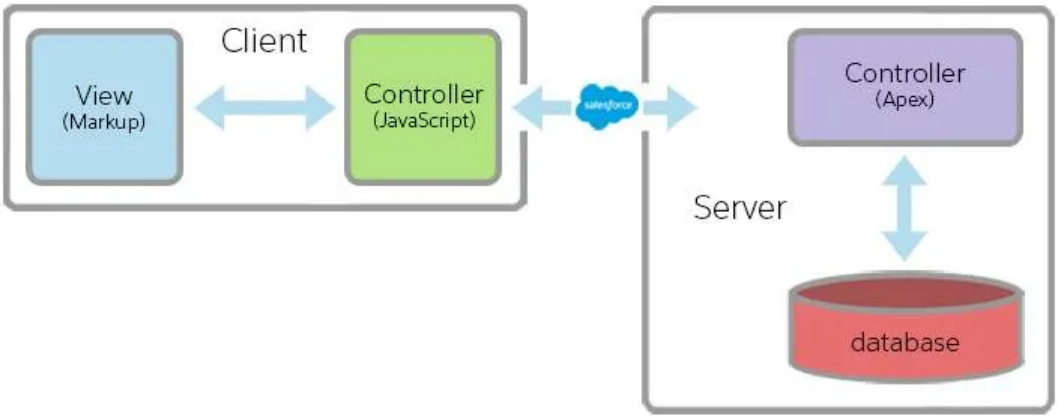
So far everything we've done has been strictly client-side. We're not saving our expenses back to Salesforce yet. Create a few expenses and then hit reload and what happens? That's right, all the expenses disappear. Woohoo, free money!

Except, Accounting just called and, well, they're kind of humorless about that kind of thing. And, actually, don't we want to be reimbursed for those expenses, which are otherwise coming out of *our* pocket...? Yikes! Saving our data to Salesforce is a P0 bug for sure!

Kidding aside, it's finally time to add server-side controllers to our app. We've been holding this back while we got the basics down. Now that you're ready for it, let's dive in!

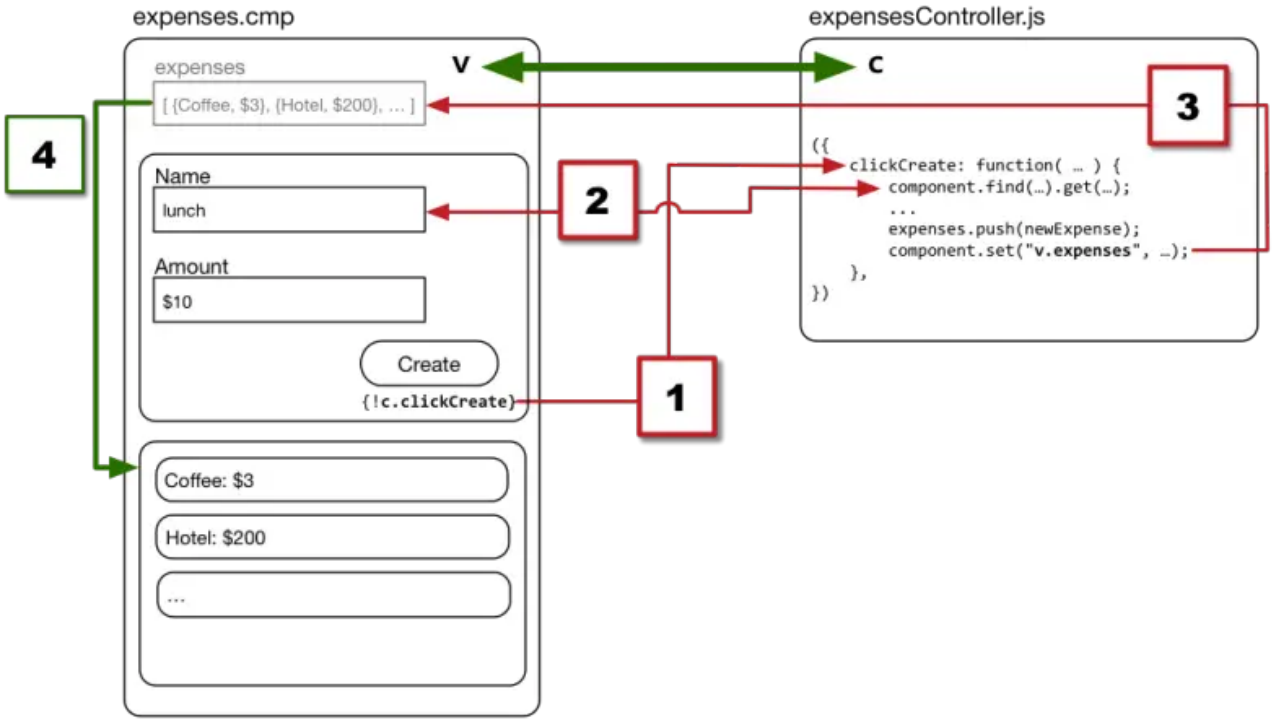
Into some pictures, that is. Let's make sure we know where we're headed, and that our gas tank is full, before we hit the road.

First, let's revisit the first diagram we saw in this module, a (very) high level look at the architecture of Lightning Components apps.



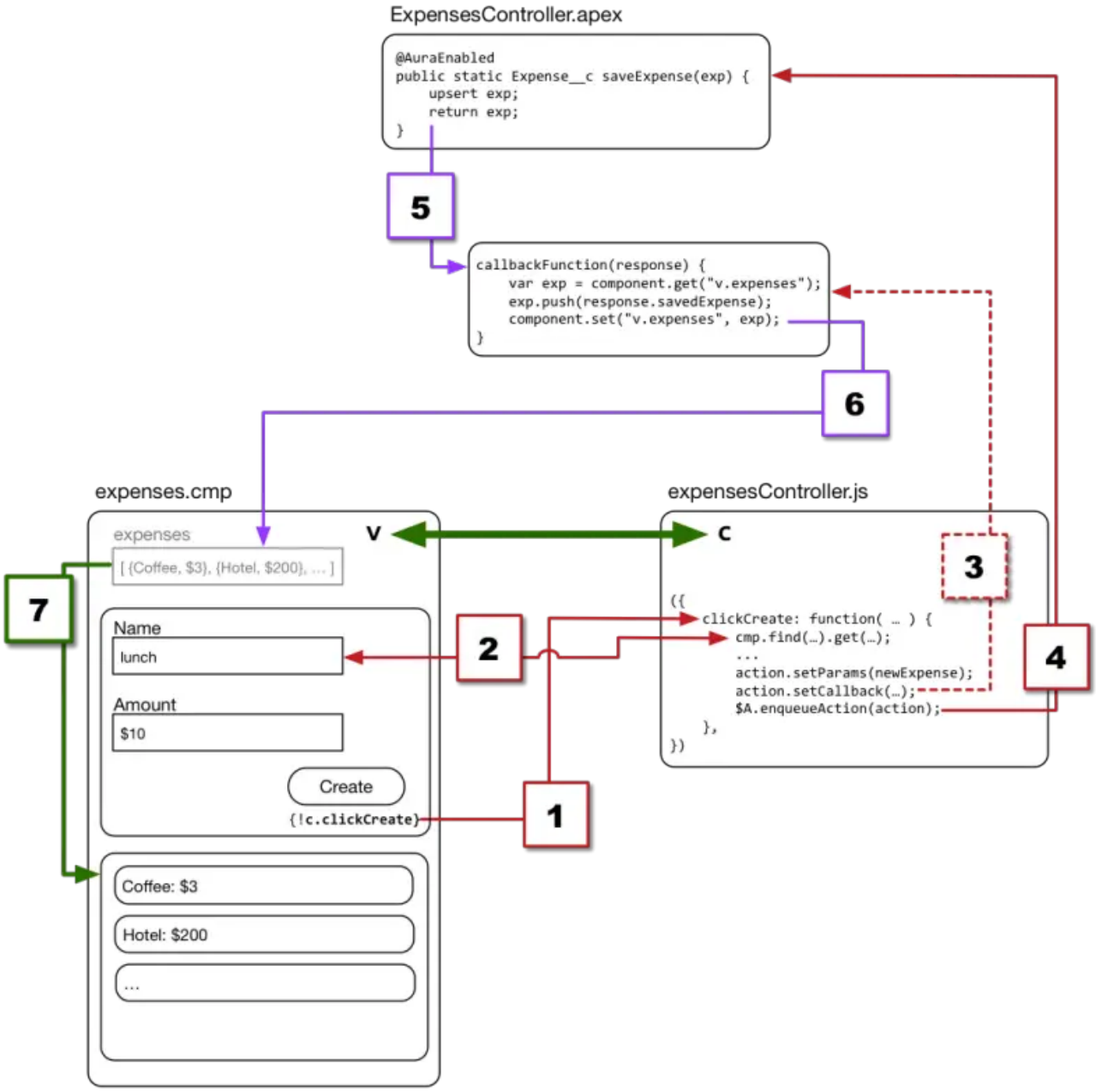
Up to now, everything we've looked at has been on the client side of this picture. (And note we simplified by merging controllers and helpers here.) Although we've referenced a custom object type, Expense__c, which is defined on the server side, we've never actually touched the server directly.

Remember how we talked about wiring together different elements to create a complete circuit? The expenses form that we built in the last unit might look something like this.



The circuit starts with the **Create** button, which is wired to the `clickCreate` action handler (1). When the action handler runs, it gets values out of the form fields (2) and then adds a new expense to the `expenses` array (3). When the array is updated via `set`, it triggers the automatic rerendering of the list of expenses (4), completing the circuit. Simple, right?

Well, when we wire in server-side access, the diagram gets a bit more complicated. More arrows, more colors, more numbers! (We'll hold off on explaining all of it for the moment.)



What's more, this circuit doesn't have the same smooth, synchronous flow of control. Server calls are expensive, and can take a bit of time. Milliseconds when things are good, and long seconds when the network is congested. You don't want apps to be locked up while waiting for server responses.

The solution to staying responsive while waiting is that server responses are handled *asynchronously*. What this means is, when you click the Create Expense button, your client-side controller fires off a server request and then keeps processing. It not only doesn't wait for the server, it forgets it ever made the request!

Then, when the response comes back from the server, code that was packaged up with the request, called a *callback function*, runs and handles the response, including updating client-side data and the user interface.

If you're an experienced JavaScript programmer, asynchronous execution and callback functions are probably your bread and butter. If you haven't worked with them before, this is going to be new, and maybe pretty different. It's also *really* cool.

Querying for Data from Salesforce

We're going to start with reading data from Salesforce, to load the list of existing expenses when the Expenses app starts up.

Note

If you haven't already created a few real expense records in Salesforce, now would be a good time. Otherwise, after implementing what follows, you might spend time debugging why nothing is loading, when there's actually just nothing to load. Your humble author falls for this one All. The. Time.

The first step is to create your Apex controller. Apex controllers contain remote methods your Lightning components can call. In this case, to query for and receive expenses data from Salesforce.

Let's take a look at a simplified version of the code. In the Developer Console, create a new Apex class named "ExpensesController" and paste in the following code.

```
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];
    }
}
```

Copy

We'll go into Apex controllers in some depth in the next section, but for now this is really a very straightforward Apex method. It runs a SOQL query and returns the results. There are only two specific things that make this method available to your Lightning Components code.

- 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.

If these requirements remind you of remote methods for Visualforce's JavaScript remoting feature, that's not a coincidence. The requirements are the same, because the architecture is very similar at key points.

One other thing worth noting is that the method doesn't do anything special to package the data for Lightning Components. It just returns the SOQL query results directly. The Lightning Components framework handles all the marshalling/unmarshalling work involved in most situations. Nice!

Loading Data from Salesforce

The next step is to wire up the `expenses` component to the server-side Apex controller. This is so easy it might make you giddy. Change the opening `<aura:component>` tag of the expenses component to point at the

Apex controller, like this.

```
<aura:component controller="ExpensesController">
```

Copy

The new part is highlighted in bold and, yes, it really is that simple.

However, pointing to the Apex controller doesn't actually load any data, or call the remote method. Like the auto-wiring between the component and (client-side) controller, this pointing simply lets these two pieces "know about" each other. This "knowing" even takes the same form, another value provider, which we'll see in a moment. But the auto-wiring only goes so far. It remains **our** job to complete the circuit.

In this case, completing the circuit means the following.

1. When the expenses component is loaded,
2. Query Salesforce for existing expense records, and
3. Add those records to the `expenses` component attribute.

We'll take each of those in turn. The first item, triggering behavior when the `expenses` component is first loaded, requires us to write an *init handler*. That's just a shorthand term for an action handler that's wired to a component's `init` event, which happens when the component is first created.

The wiring you need for this is a single line of markup. Add the following to the `expenses` component, right below the component's attribute definitions.

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

Copy

The `<aura:handler>` tag is how you say that a component can, well, *handle* a specific event. In this case, we're saying that we'll handle the `init` event, and that we'll handle it with the `doInit` action handler in our controller. (Setting `value="{!this}"` marks this as a "value event." What this means is too complex to go into here. Just know that you should always add this attribute-value pair to an `init` event.)

Calling Server-Side Controller Methods

One step down, two to go. Both of the remaining steps take place in the `doInit` action handler, so let's look at it. Add the following code to the `expense` component's controller.

```
    // Load expenses from Salesforce
    doInit: function(component, event, helper) {
        // Create the action
        var action = component.get("c.getExpenses");
        // Add callback behavior for when response is received
        action.setCallback(this, function(response) {
            var 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);
    },
```

Copy

Before you feel lost with all the new things here, please notice that this is just another action handler. It's formatted the same, and the function signature is the same. We're in familiar territory.

That said, every line of code after the function signature is new. We'll look at all of them in a moment, but here's the outline of what this code does:

1. Create a remote method call.
2. Set up what should happen when the remote method call returns.
3. Queue up the remote method call.

That sounds pretty simple, right? Maybe the structure or specifics of the code is new, but the basic requirements of what needs to happen are once again familiar.

Does it sound like we're being really encouraging? Like, maybe we're trying to coach you through a rough patch? Well, we need to talk about something tough. The issue appears in the first line of code in the function.

```
        var action = component.get("c.getExpenses");
```

Copy

This line of code creates our remote method call, or remote action. And at first the `component.get()` part looks familiar. We've done that many times at this point.

Except…well, before it was "v.something" that we were getting, with `v` being the value provider for the view. Here it's "c", and yes, `c` is another value provider. And we've seen a `c` value provider before, in expressions like `press="{!c.clickCreate}"` and `action="{!c.doInit}"`.

Those expressions were in component markup, in the view. Here in the controller, the `c` value provider represents something different. It represents the remote Apex controller.

"Wait a minute. Are you telling me we have `c` the client-side controller, `c` the default namespace, and `c` the server-side controller, all in Aura components?"

Well, in a word, yes. Deep breaths.

Look, we'll be honest with you. If we had it all to do over again, we might have made some different choices. While the choices we made weren't accidents, three "c"s is definitely an opportunity for confusion. We get confused too!

But as they say, it is what it is. Forewarned is forearmed. Now you know.

Identifier	Context	Meaning

Identifier	Context	Meaning
`c.`	Component markup	Client-side controller
`c.`	Controller code	Server-side controller
`c:`	Markup	Default namespace

OK, back to our code. Before we got sidetracked, we were looking at this line.

```
        var action = component.get("c.getExpenses");
```

Copy

Where, in earlier code, `component.get("v.something")` returned to us a reference to a child component in the view (component markup), `component.get("c.whatever")` returns a reference to an action available in the controller. In this case, it returns a remote method call to our Apex controller. This is how you create a call to an `@AuraEnabled` method.

The next "line," action.setCallback(…), is a block of code that will run when the remote method call returns. Since this happens "later," let's set it aside for the moment.

The next line that actually *runs* is this one.

```
        $A.enqueueAction(action);
```

Copy

We saw `$A` briefly before, but didn't discuss it. It's a framework global variable that provides a number of important functions and services. `$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.

That sounds kind of vague. The full details are interesting, and important for advanced use of Aura components. But for now, here's what you need to know about `$A.enqueueAction(action)`.

- 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 where that block of code we set aside comes in. But before we talk about that, a little pop culture.

Server Calls, Asynchronous Execution, and Callback Functions

Carly Rae Jepsen's single "Call Me Maybe" was released in 2011, to critical and commercial success, hitting #1 in more than a dozen countries. To date it has sold more than 18 million copies worldwide and is, apparently, one of the best-selling digital singles of all time. The most memorable line, from the chorus, is "Here's my number. So call me maybe." In addition to being upbeat and dangerously catchy, it's a metaphor for how Aura components handle server calls.

Hear us out. Let's look at our action handler in pseudo-code.

```
    doInit: function(component, event, helper) {
        // Load expenses from Salesforce
        var action = component.get("c.getExpenses");
        action.setCallback(
            // Here's my number,
            // Call me maybe
        );
        $A.enqueueAction(action);
    },
```

Copy

Hmmm. Maybe we should explain the parameters to `action.setCallback()` in more detail. In the real action handler code, we call it like so.

```
        action.setCallback(this, function(response) { ... });
```

Copy

`this` is the scope in which the callback will execute; here `this` is the action handler function itself. Think of it as an address, or...maybe a **number**. The function is what gets **called** when the server response is returned. So:

```
        action.setCallback(scope, callbackFunction);
```

Copy

Here's my number. Call me maybe.

The overall effect is to create the request, package up the code for what to do when the request is done, and send it off to execute. At that point, the action handler itself stops running.

Here's another way to wrap your head around it. You might bundle your child up for school, and hand them a list of the chores you want them to do when they come home after classes. You drop them off at school, and then you go to work. While you're at work, you're doing *your work*, secure in the knowledge that your child, being a good kid, will do *the work you assigned to them* when they get back from school. You don't do that work yourself, and you don't know when, exactly, it will happen. But it does.

Here's one last way to look at it, again in pseudo-code. This version "unwraps" the callback function to show a more linear version of the action handler.

```
    // Not real code! Do not cut-and-paste!
    doInit: function(component, event, helper) {
        // Create server request
        var action = component.get("c.getExpenses");
        // Send server request
        $A.enqueueAction(action);
        // ... time passes ...
```

```
        // ...
        // ... Jeopardy theme plays ...
        // ...
        // ... at some point in the indeterminate future ...
        // Handle server response
        var state = action.response.getState();
        if (state === "SUCCESS") {
            component.set("v.expenses", action.response.getReturnValue());
        }
    },
```

Copy

We'll say it again. Asynchronous execution and callback functions are *de rigueur* for JavaScript programmers, but if you're coming from another background, it might be less familiar. Hopefully we've got it down at this point, because it's fundamental to developing apps with Lightning Components.

Handling the Server Response

Now that we've got the structure of creating a server request down, let's look at the details of how our callback function actually handles the response. Here's just the callback function.

```
    function(response) {
        var state = response.getState();
        if (state === "SUCCESS") {
            component.set("v.expenses", response.getReturnValue());
        }
    }
```

Copy

Callback functions take a single parameter, `response`, which is an opaque object that provides the returned data, if any, and various details about the status of the request.

In this specific callback function, we do the following.

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.

You probably have a few questions, such as:

- What happens if the response state isn't SUCCESS?
- What happens if the response never comes? (Call me *maybe*.)
- How can we just assign the response data to our component attribute?

The answers to the first two are that, unfortunately, we're not going to cover those possibilities in this module. They're certainly things you need to know about and consider in your real world apps, but we just don't have the space.

The last question is the most relevant here, but it's also the easiest to answer. We defined a data type for the expenses attribute.

```
<aura:attribute name="expenses" type="Expense__c[]"/>
```

Copy

And our server-side controller action has a method signature that defines its return data type.

```
public static List<Expense__c> getExpenses() { ... }
```

Copy

The types match, and so we can just assign one to the other. Aura components handle all the details. You can certainly do your own processing of the results, and turn it into other data within your app. But if you design your server-side actions right, you don't necessarily have to.

OK, that was a lot of different ways to look at a dozen lines of code. Here's the question: Have you tried your version of our app with it yet? Because we're done with the loading expenses from Salesforce part. Go reload the app, and see if the expenses you entered in Salesforce show up!

Apex Controllers for Aura Components

Before we take the next step in developing the app, let's dive a little deeper into that Apex controller. Here's a look at the next version, which we'll need to handle creating new records, as well as updating the Reimbursed? checkbox on existing records.

```
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;
    }
}
```

Copy

The earlier version promised a stern lecture, and that's coming. But first, let's focus on the details of this minimal version.

First, we've added only one new `@AuraEnabled` method, `saveExpense()`. It takes an Expense (`Expense__c`) object and upserts it. This allows us to use it to both create new records and to update existing records.

Next, notice that we created the class with the `with sharing` keywords. This automatically applies your org's sharing rules to the records that are available via these methods. For example, users would normally only see their own expense records. Salesforce handles all the complicated SOQL rules for you, automatically, behind the scenes.

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];
    }
```

Copy

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) {
        var action = component.get("c.saveExpense");
        action.setParams({
            "expense": expense
        });
        action.setCallback(this, function(response){
            var state = response.getState();
            if (state === "SUCCESS") {
                var expenses = component.get("v.expenses");
                expenses.push(response.getReturnValue());
                component.set("v.expenses", expenses);
            }
        });
        $A.enqueueAction(action);
    },
```

Copy

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"/>
```

Copy

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.

```
var validExpense = component.find('expenseform').reduce(function (validSoFar, inputCmp) {
    // Displays error messages for invalid fields
    inputCmp.showHelpMessageIfInvalid();
    return validSoFar && inputCmp.get('v.validity').valid;
}, true);
```

Copy

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
- Creating Server-Side Logic with Controllers
- Calling a Server-Side Action
- Working with Salesforce Records
- Saving Records
- CRUD and Field-Level Security (FLS)
- Returning Errors from an Apex Server-Side Controller
- Queuing of Server-Side Actions
- Testing Your Apex Code

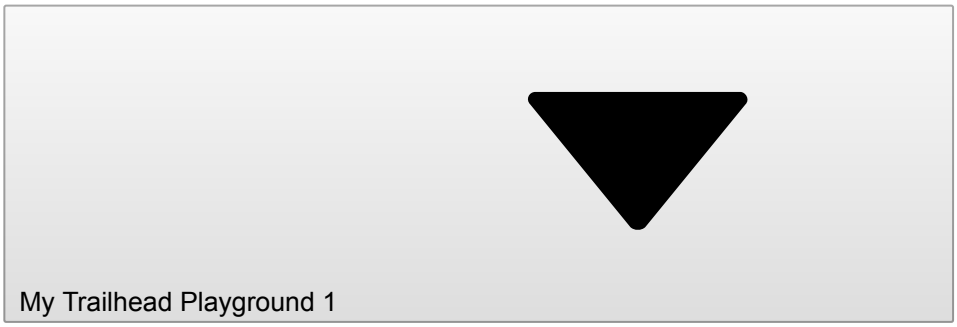
Get Ready

You'll be completing this challenge in your own personal Salesforce environment. Choose from the dropdown menu, then click **Launch** to get started. If you use Trailhead in a language other than English, set the language of your Trailhead Playground to English before you attempt this challenge. Want to find out more about using hands-on orgs for Trailhead learning? Check out the Trailhead Playground Management module.

Your Challenge

Save and Load Records with a Server-Side Controller
Persist your records to the database using a server-side controller. The **campingList** component loads existing records when it starts up and saves records to the database when the form is submitted.

- Create a **CampingListController** Apex class with a **getItems** method and **saveItem** method.
- Add a **doInit** initialization handler that loads existing records from the database when the component starts up.
- Modify the JavaScript controller to use a **createItem** method in the helper to save records to the database from a valid form submission. The new items are added to the controller's **items** value provider.



Launch

Check challenge to earn 500 points