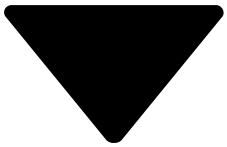
📣 Attention Salesforce Certified Trailblazers! Maintain your credentials and link your Trailhead and Webassessor accounts by December 6th. Learn more.
close

1. Aura Components Basics



2. Connect Components with Events

Connect Components with Events

Learning Objectives

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

- Define custom events for your apps.
- Create and fire events from a component controller.
- Create action handlers to catch and handle events that other components send.
- Refactor a big component into smaller components.

Connect Components with Events

In this unit we're going to tackle the last piece of unfinished functionality in our little expenses app: the Reimbursed? checkbox. You're probably thinking that implementing a checkbox would be a short topic. We could certainly take some shortcuts, and make it a *very* short topic.

But this unit, in addition to making the checkbox work, is about removing all of the shortcuts we've already taken. We're going to spend this unit "Doing It Right." Which, in a few places, means refactoring work we did earlier.

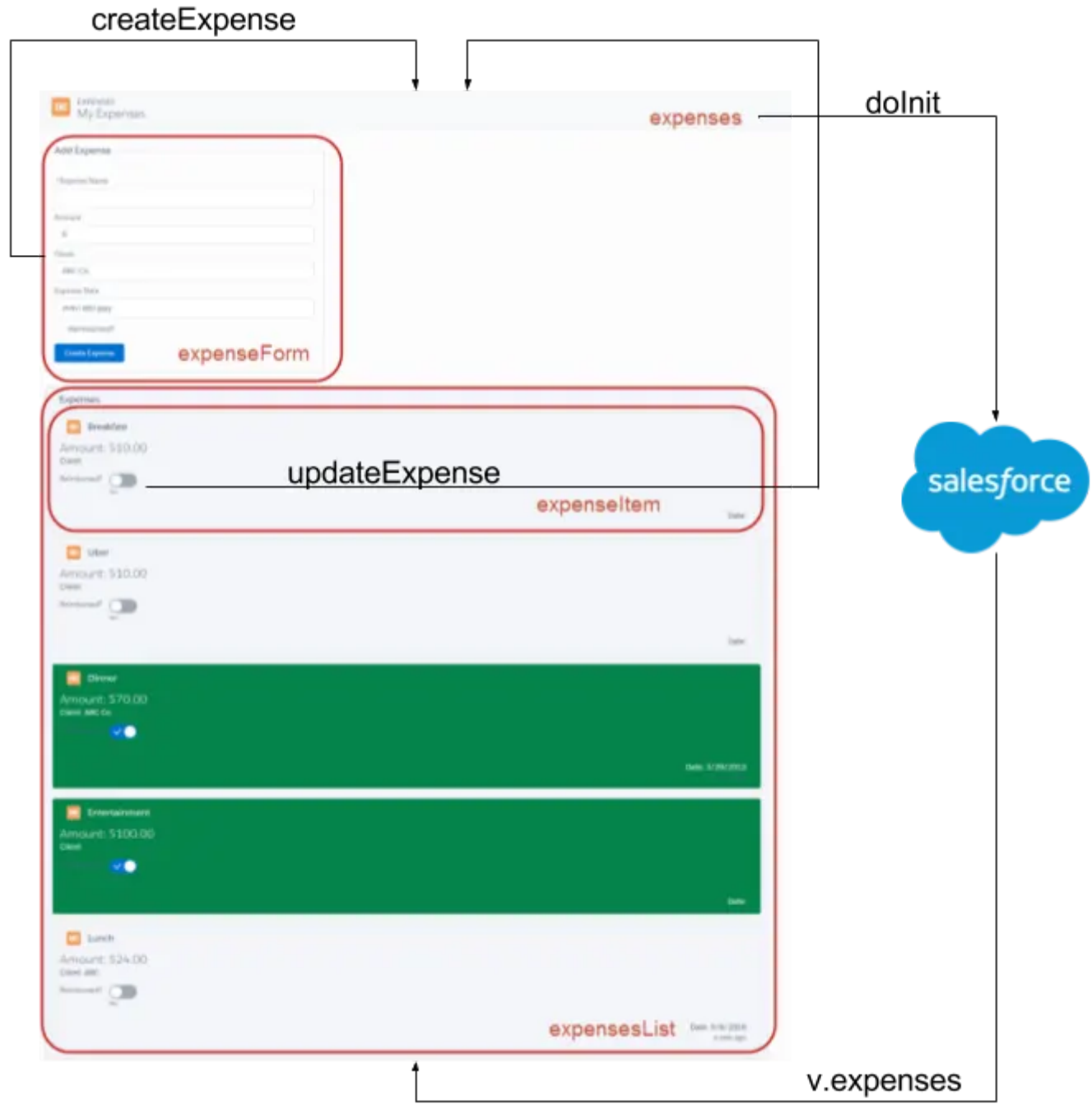
Before we start on that, let's first talk about the shortcuts we took, the Right Way, and why the Right Way is (a little bit) harder, but also better.

Composition and Decomposition

If you take a look at our little expenses app in source code form, and list the separate code artifacts, you'll come up with something like the following.

- expenses component
  - expenses.cmp
  - expensesController.js
  - expensesHelper.js
- expensesList component
  - expensesList.cmp
- expenseItem component
  - expenseItem.cmp
- ExpensesController (server-side)
  - ExpensesController.apex

Here's how everything comes together, with the `createExpense` and `updateExpense` events you're wiring up later.



But, if you look at the app on screen, what do you see? What you should see, and what you'll eventually see everywhere you look, is that the app breaks down into many more components. You'll see that you can *decompose* our app further, into smaller pieces, than we've done so far. At the very least, we hope you see that the Add Expense form really should be its own, separate component. (That's why we drew a box around it in the user interface!)

Why didn't we make that form a separate component? Not doing that is by far the biggest shortcut we took over the course of this module. It's worse than the hack we called "disgusting," in terms of software design. The right way to build a Lightning Components app is to create independent components, and then *compose* them together to build new, higher level features. Why didn't we take that approach?

We took the shortcut, we kept the Add Expense form inside the main `expenses` component, because it kept the main `expenses` array component attribute and the controller code that affected it in the same component. We wanted the `createExpense()` helper function to be able to touch the `expenses` array directly. If we'd moved the Add Expense form into a separate component, that wouldn't have been possible.

Why not? We covered the reason briefly very early on, but we want to really hammer on it now. Lightning components are supposed to be self-contained. They are stand-alone elements that encapsulate all of their essential functionality. A component is not allowed to reach into another component, even a child component, and alter its internals.

There are two principal ways to interact with or affect another component. The first way is one we've seen and done quite a bit of already: setting attributes on the component's tag. A component's public attributes constitute one part of its API.

The second way to interact with a component is through *events*. Like attributes, components declare the events they send out, and the events they can handle. Like attributes, these public events constitute a part of the component's public API. We've actually used and handled events already, but the events have been hiding behind some convenience features. In this unit, we'll drag events out into the light, and create a few of our own.

The Wiring-a-Circuit Metaphor, Yet Again

These two mechanisms—attributes and events—are the API "sockets," the ways you connect components together to form complete circuits. Events are also, behind the scenes, the electrons flowing through that circuit. But that's only one way that events are different from attributes.

When you set the `onclick` attribute on a `<lightning:button>` to an action handler in a component's controller, you create a direct relationship between those two components. They are linked, and while they're using public APIs to remain independent of each other, they're still coupled.

Events are different. Components don't send events *to* another component. That's not how events work. Components broadcast events of a particular type. If there's a component that responds to that type of event, and if that component "hears" your event, then it will act on it.

You can think of the difference between attributes and events as the difference between wired circuits and wire**less** circuits. And we're not talking wireless phones here. One component doesn't get the "number" for another component and call it up. That would be an attribute. No, events are like wireless broadcasts. Your component gets on the radio, and sends out a message. Is there anyone out there with their radio set turned on, and tuned to the right frequency? Your component has no way of knowing—so you should write your components in such a way that it's OK if no one hears the events they broadcast. (That is, things might not work, but nothing should crash.)

Sending an Event from a Component

OK, enough theory, let's do something specific with our app, and see how events work in code. We will start by implementing the Reimbursed? checkbox. Then we'll take what we learned doing that, and use it to refactor the Add Expense form into its own component, the way the Great Engineer intended.

First, let's focus on the click handler on the `<lightning:input>` for the `Reimbursed__c` field.

```
<lightning:input type="toggle"
            label="Reimbursed?"
            name="reimbursed"
            class="slds-p-around--small"
            checked="{!v.expense.Reimbursed__c}"
            messageToggleActive="Yes"
```

```
            messageToggleInactive="No"
            onchange="{!c.clickReimbursed}"/>
```

Copy

Before we dive into the click handler, let's take a step back to catch a glimpse on what `<lightning:input>` has to offer. `type="toggle"` is really a checkbox with a toggle switch design. `class` enables you to apply custom CSS styling or use SLDS utilities. `messageToggleActive` and `messageToggleInactive` provides a custom label for the checked and unchecked positions. These handy attributes are only several of many others on `<lightning:input>`. Finally, the `onchange` attribute of `<lightning:input>` gives us an easy way to wire the toggle switch to an action handler that updates the record when you slide right (checked) or slide left (unchecked).

Now, let's think about what should happen when it's checked or unchecked. Based on the code we wrote to create a new expense, updating an expense is probably something like this.

1. Get the `expense` item that changed.
2. Create a server action to update the underlying expense record.
3. Package `expense` into the action.
4. Set up a callback to handle the response.
5. Fire the action, sending the request to the server.
6. When the response comes and the callback runs, update the `expenses` attribute.

Um, what `expenses` attribute? Look at our component markup again. No `expenses`, just a singular `expense`. Hmmm, right, this component is just for a single item. There's an `expenses` attribute on the `expensesList` component…but that's not even the "real" `expenses`. The real one is a component attribute in the top-level `expenses` component. Hmmm.

Is there a `component.get("v.parent")`? Or would it have to be `component.get("v.parent").get("v.parent")` — something that would let us get a reference to our parent's parent, so we can set `expenses` there?

Stop. Right. There.

Components do not reach into other components and set values on them. There's no way to say "Hey grandparent, I'm gonna update `expenses`." Components keep their hands to themselves. When a component wants an ancestor component to make a change to something, it **asks**. Nicely. By sending an event.

Here's the cool part. Sending an event looks almost the same as handling the update directly. Here's the code for the `clickReimbursed` handler.

```
({
    clickReimbursed: function(component, event, helper) {
        var expense = component.get("v.expense");
        var updateEvent = component.getEvent("updateExpense");
        updateEvent.setParams({ "expense": expense });
        updateEvent.fire();
    }
})
```

Copy

Whoa. That's pretty simple! And it does look kind of like what we envisioned above. The preceding code for `clickReimbursed` does the following:

1. Gets the `expense` that changed.

2. Creates an event named `updateExpense`.
3. Packages `expense` into the event.
4. Fires the event.

The callback stuff is missing, but otherwise this is familiar. But…what's going to handle calling the server, and the server response, and update the main `expenses` array attribute? And how do we know about this `updateExpense` event, anyway?

`updateExpense` is a custom event, that is, an event we write ourselves. You can tell because, unlike getting a server action, we use `component.getEvent()` instead of `component.get()`. Also, what we are getting doesn't have a value provider, just a name. We'll define this event in just a moment.

As for what's going to handle calling the server and handling the response, let's talk about it. We could implement the server request and handle the response right here in the `expenseItem` component. Then we'd send an event to just rerender things that depend on the `expenses` array. That would be a perfectly valid design choice, and would keep the `expenseItem` component totally self-contained, which is desirable.

However, as we'll see, the code for creating a new expense and the code for updating an existing expense are very similar, enough so that we'd prefer to avoid duplicate code. So, the design choice we've made is to send an `updateExpense` event, which the main `expenses` component will handle. Later, when we refactor our form, we'll do the same for creating a new expense.

By having all child components delegate responsibility for handling server requests and for managing the `expenses` array attribute, we're breaking encapsulation a bit. But, if you think of these child components as the internal implementation details of the `expenses` component, that's OK. The main `expenses` component *is* self-contained.

You have a choice: consolidation of critical logic, or encapsulation. You'll make trade-offs in Aura components just like you make trade-offs in any software design. Just make sure you document the details.

Defining an Event

The first thing we'll do is define our custom event. In the Developer Console, select **File** | **New** | **Lightning Event**, and name the event "expensesItemUpdate". Replace the default contents with the following markup.

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

Copy

There are two types of events, component and application. Here we're using a component event, because we want an ancestor component to catch and handle the event. An ancestor is a component "above" this one in the component hierarchy. If we wanted a "general broadcast" kind of event, where *any* component could receive it, we'd use an application event instead.

The full differences and correct usage of application vs. component events isn't something we're able to get into here. It's a more advanced topic, and there are enough complicated details that it's a distraction from our purpose in this module. When you're ready for more, the Resources will help you out.

The other thing to notice about the event is how compact the definition is. We named the event when it was created, `expensesItemUpdate`, and its markup is a beginning and ending `<aura:event>` tag, and one `<aura:attribute>` tag. An event's attributes describe the payload it can carry. In the `clickReimbursed` action

handler, we set the payload with a call to `setParams()`. Here in the event definition, we see how the event parameter is defined, and that there are no other valid parameters.

And that's pretty much all there is to defining events. You don't add implementation or behavior details to events themselves. They're just packages. In fact, some events don't have any parameters at all. They're just messages. "This happened!" All of the behavior about what to do if "this" happens is defined in the components that send and receive the event.

Sending an Event

We already looked at how to actually *fire* an event, in the `clickReimbursed` action handler. But for that to work, we need to do one last thing, and that's register the event. Add this line of markup to the `expenseItem` component, right below its attribute definitions.

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

Copy

This markup says that our component fires an event, named "updateExpense", of type "c:expensesItemUpdate". But, wasn't "expensesItemUpdate" the *name* of the event when we defined it? And what happened to component or application event *types*?

You're right to think it's a little confusing—it really is a bit of a switch-a-roo. It might help to think of "application" and "component" as Aura components *framework* event types, while the types that come from the names of events you define are custom event types, or event *structure* types. That is, when you define an event, you define a package format. When you register to send an event, you declare what format it uses.

The process of defining and registering an event might still seem a bit weird, so let's look ahead a bit. Here in `expenseItem`, we're going to send an event named `updateExpense`. Later in `expenseForm`, we're going to send an event named `createExpense`. Both of these events need to include an expense to be saved to the server. And so they both use the `c:expensesItemUpdate` event type, or package format, to send their events.

On the receiving end, our main `expenses` component is going to register to handle *both* of these events. Although the server call ends up being the same, the user interface updates are slightly different. So how does `expenses` know whether to create or update the expense in the `c:expensesItemUpdate` package? By the *name* of the event being sent.

Understanding the distinction here, and how one event can be used for multiple purposes, is a light bulb moment in learning Lightning Components. If you haven't had that moment quite yet, you'll have it when you look at the rest of the code.

Before we move on to handling events, let's summarize what it takes to send them.

1. Define a custom event by creating a Lightning Event, giving it a name and attributes.
2. Register your component to send these events, by choosing a custom event type and giving this specific use of that type a name.
3. Fire the event in your controller (or helper) code by:
    a. Using `component.getEvent()` to create a specific event instance.
    b. Sending the event with `fire()`.

If you went ahead and implemented all of the code we just looked at, you can test it out. Reload your app, and toggle a Reimbursed? checkbox a few times. If you missed a step, you'll get an error, and you should recheck your work. If you did everything right…hey, wait, the expense changes color to show its Reimbursed? status, just as expected!

This behavior was present before we even started this unit. That's the effect of the `<lightning:input>` component having `value="{!v.expense.Reimbursed__c}"` set. When you toggle the switch, the *local* version of the `expense` is updated. But that change isn't being sent up to the server. If you look at the expense record in Salesforce, or reload the app, you won't see the change.

Why not? We've only done *half* of the work to create a complete circuit for our event. We have to finish wiring the circuit by creating the event handler on the other side. That event handler will take care of sending the change to the server, and making the update durable.

Handling an Event

Enabling the `expenseItem` component to send an event required three steps. Enabling the `expenses` component to receive and handle these events requires three parallel steps.

1. Define a custom event. We've already done this, because `expenseItem` is sending the same custom event that `expenses` is receiving.
2. Register the component to handle the event. This maps the event to an action handler.
3. Actually handle the event in an action handler.

Since we've already done step 1, let's immediately turn to step 2, and register `expenses` to receive and handle the `updateExpense` event. Like registering to send an event, registering to handle one is a single line of markup, which you should add to the `expenses` component right after the init handler.

```
    <aura:handler name="updateExpense" event="c:expensesItemUpdate"
      action="{!c.handleUpdateExpense}"/>
```

Copy

Like the init handler, this uses the `<aura:handler>` tag, and has an `action` attribute that sets the controller action handler for the event. Like when you registered the event in `expenseItem`, here you set the name and type of the event—though note the use of the much more sensibly named `event` attribute for the type.

In other words, there's not much here you haven't seen before. What's new, and specific to handling custom events, is the combination of the attributes, and knowing how this receiver "socket" in `expenses` matches up with the sender "socket" in `expenseItem`.

This completes the wiring part of making this work. All that's left is to actually write the action handler!

We'll start with the `handleUpdateExpense` action handler. Here's the code, and be sure to put it riiiiight under the `clickCreate` action handler.

```
    handleUpdateExpense: function(component, event, helper) {
        var updatedExp = event.getParam("expense");
        helper.updateExpense(component, updatedExp);
    }
```

Copy

Huh. That's interesting. Except for the form validation check and the specific helper function we're delegating the work to, it looks like this action handler is the same as `handleCreateExpense`.

And now let's add the `updateExpense` helper function. As we did with the action handler, make sure you put this code right below the `createExpense` helper function.

```
    updateExpense: 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") {
                // do nothing!
            }
        });
        $A.enqueueAction(action);
    },
```

Copy

Two things you should notice right off the bat. First, except for the callback specifics, the `updateExpense` helper method is identical to the `createExpense` helper method. That smells like opportunity.

Second, about those callback specifics. What gives? How can the right thing to do be *nothing*?

Think about it for a moment. Earlier, when testing sending the event (if not before), we saw that the `expenseItem` component's color changed in response to toggling the Reimbursed? checkbox. Remember the explanation? The local copy of the expense record is *already updated!* So, at least for the moment, when the server tells us it was successful at updating its version, we don't have to do anything.

Note that this code only handles the case where the server is *successful* at updating the expense record. We'd definitely have some work to do if there was an error. Say, Accounting flagged this expense as non-reimbursable, making it impossible to set this field to `true`. But that, as they say, is a lesson for another day.

Refactor the Helper Functions

Let's go back to that opportunity we saw to factor out some common code. The two helper functions are identical except for the callback. So, let's make a new, more generalized function that takes the callback as a parameter.

```
    saveExpense: function(component, expense, callback) {
        var action = component.get("c.saveExpense");
        action.setParams({
            "expense": expense
        });
        if (callback) {
            action.setCallback(this, callback);
        }
        $A.enqueueAction(action);
    },
```

Copy

The `callback` parameter is optional. If it's there, we'll pass it along to `action`. Simple. And now we can reduce our event-specific helper functions to the following code.

```
    createExpense: function(component, expense) {
        this.saveExpense(component, expense, function(response){
            var state = response.getState();
            if (state === "SUCCESS") {
                var expenses = component.get("v.expenses");
                expenses.push(response.getReturnValue());
                component.set("v.expenses", expenses);
            }
        });
    },
    updateExpense: function(component, expense) {
        this.saveExpense(component, expense);
    },
```

Copy

`createExpense` is only a little shorter, but it's exclusively focused on what to do when the response comes back (the callback). And wow, `updateExpense` is a one-liner!

Refactor the Add Expense Form

That little refactoring exercise was so satisfying, and using events was so (we're sorry) electrifying, let's do it again, but bigger. *More cowbell!*

This next task involves extracting the Add Expense form from the expenses component, and moving it to its own, new component. Extracting the form markup is easy enough, a simple copy-and-paste exercise. But what else moves with it? Before we start moving pieces around willy-nilly, let's think about what moves and what stays.

In the current design, the form's action handler, `clickCreate`, handles input validation, sending the request to the server, and updating local state and user interface elements. The form will still need an action handler, and should probably still handle form validation. But we'll plan on having the rest stay behind, because we're keeping our server request logic consolidated in the `expenses` component.

So there's a little (but only a little!) teasing apart to do there. Our plan, then, is to start by moving the form markup, and then move as little as possible after that to make it work correctly. We'll refactor both components to communicate via events, instead of via direct access to the `expenses` array component attribute.

Let's get started!

In the main `expenses` component, select everything between the two `<!-- CREATE NEW EXPENSE -->` comments, including the beginning and ending comments themselves. Cut it to your clipboard. (Yes, *cut*. We're committed.)

Create a new Aura component, and name it "expenseForm". Paste the copied Add Expense form markup into the new component, between the `<aura:component>` tags.

Back to the `expenses` component. Add the new `expenseForm` component to the markup. That section of `expenses` should look like this.

```
    <!-- NEW EXPENSE FORM -->
    <lightning:layout >
        <lightning:layoutItem padding="around-small" size="6">
            <c:expenseForm/>
```

```
        </lightning:layoutItem>
    </lightning:layout>
```

Copy

At this point, you can reload your app to see the changes. There should be no *visible* changes. But, unsurprisingly, the Create Expense button no longer works.

Let's get quickly through the rest of the moving things around part.

Next, move the `newExpense` attribute from the `expenses` component to the `expenseForm` component markup. This is used for the form fields, so it needs to be in the form component. It moves over with no changes required, so just cut from one and paste in the other.

In the `expenseForm` component, create the controller and helper resources.

Move the `clickCreate` action handler from the `expenses` controller to the `expenseForm` controller. The button is in the form component, and so the action handler for the button needs to be there, too. Believe it or not, this also needs no changes. (You might begin sensing a theme here.)

Now we need to make a couple of actual changes. But these will be familiar, because we're just adding event sending, which we did before for `expenseItem`. `expenseItem`, you'll recall, also sends an event with an `expense` payload, which is handled by the `expenses` component.

In the `expenseForm` helper, create the `createExpense` function.

```
    createExpense: function(component, newExpense) {
        var createEvent = component.getEvent("createExpense");
        createEvent.setParams({ "expense": newExpense });
        createEvent.fire();
    },
```

Copy

This looks very much like the `clickReimbursed` action handler in `expenseItem`.

If a component is going to send an event, it needs to register the event. Add the following to the `expenseForm` component markup, just below the `newExpense` attribute.

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

Copy

At this point, we've done all the work to implement the `expenseForm` component. You should be able to reload the app, and the form now "works" in the sense that there are no errors, and you should see the appropriate form messages when you enter invalid data. If you're using the Salesforce Lightning Inspector, you can even see that the `expensesItemUpdate` event is being fired. All that's left is to handle it.

Before we handle the event, please do notice how easy this refactoring was. Most of the code didn't change. There's a total of six lines of new code and markup in the preceding steps. It's unfamiliar to do this work today, but do it a few times, and you realize that you're just moving a bit of code around.

OK, let's finish this. The `expenseForm` fires the `createExpense` event, but we also need the `expenses` component to catch it. First we register the `createExpense` event handler, and wire it to the `handleCreateExpense` action handler. Once again, this is a single line of markup. Add this line right above or below the `updateExpense` event handler.

```
    <aura:handler name="createExpense" event="c:expensesItemUpdate"
        action="{!c.handleCreateExpense}"/>
```

Copy

Finally, for the last step, create the `handleCreateExpense` action handler in the `expenses` controller. Add this code right above or below the handleUpdateExpense action handler.

```
    handleCreateExpense: function(component, event, helper) {
        var newExpense = event.getParam("expense");
        helper.createExpense(component, newExpense);
    },
```

Copy

Yep, that simple. All of the work is delegated to the `createExpense` helper function, and that didn't move or change. Our `handleCreateExpense` action handler is just there to wire the right things together.

And with that, we're finished showing how to loosely couple components using events. Create and fire the event in one component, catch and handle it in another. Wireless circuits!

Bonus Lesson—Minor Visual Improvements

Before we head off into the sunset, or rather, the challenge, here is a modest visual improvement.

We'd like to improve the layout of our app a little bit, by adding a few container components. This last bit also gives you an opportunity to see the full `expense` component after all our changes. In the `expense` component, replace the `expensesList` markup with the following.

```
<lightning:layout>
    <lightning:layoutItem padding="around-small" size="6">
        <c:expensesList expenses="{!v.expenses}"/>
    </lightning:layoutItem>
    <lightning:layoutItem padding="around-small" size="6">
        Put something cool here
    </lightning:layoutItem>
</lightning:layout>
```

Copy

The effects of the changes are to add some margins and padding, and make the expenses list more narrow. The layout leaves room to put something over there on the right. In the next unit, we'll suggest a couple of exercises you could do on your own.

Resources

- Communicating with Events
- Actions and Events
- Handling Events with Client-Side Controllers
- Event Handling in Base Lightning Components
- Invoking Actions on Component Initialization
- Detecting Data Changes

- Component Events
- Handling Component Events
- Dynamically Showing or Hiding Markup
- Application Events
- Handling Application Events
- Event Handling Lifecycle
- Advanced Events Example

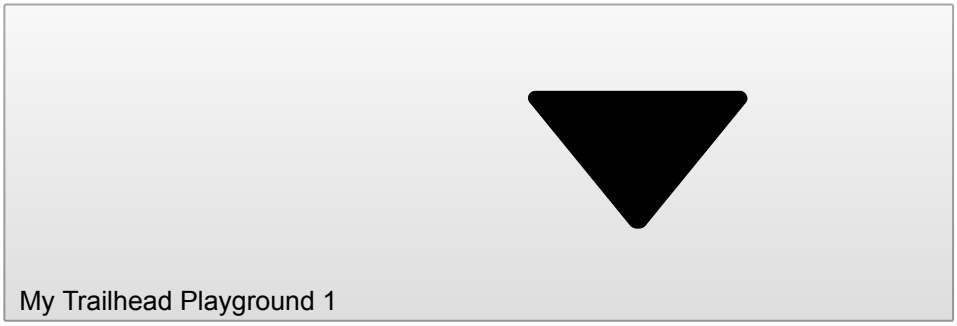
Get Ready

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

Refactor Components and Communicate with Events
Refactor the input form for camping list items into its own component and communicate with component events.

- Replace the HTML form in the campingList component with a new **campingListForm** component that calls the **clickCreateItem** JavaScript controller action when clicked.
- The campingList component listens for a **c:addItemEvent** event and executes the action **handleAddItem** in the JavaScript controller. The **handleAdditem** method saves the record to the database and adds the record to the **items** value provider.
- The **addItemEvent** event is of type **component** and has a Camping_Item__c type attribute named **item**.
- The campingListForm registers an **addItem** event of type **c:addItemEvent**.
- The campingListFormController JavaScript controller calls the helper's **createItem** method if the form is valid.
- The campingListFormHelper JavaScript helper creates an **addItem** event with the item to be added and then fires the event. It then resets the **newItem** value provider with a blank sObjectType of type Camping_Item__c.



Launch

Check challenge to earn 500 points