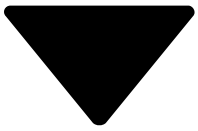
1. Lightning Data Service Basics for Aura Components



2. Manipulate Records with force:recordData

Manipulate Records with force:recordData

Learning Objectives

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

- Use Lightning Data Service for Aura Components to create, read, upload, and delete records.
- Build a component that uses `force:recordData`.
- Explain how Lightning Data Service caches records.

Use the force... recordData Tag

In the last unit, we covered the nifty performance upgrades and quality-of-life features that Lightning Data Service provides. Now let's learn how to use them.

Remember, `force:recordData` doesn't inherently include any UI elements. The `force:recordData` tag is just the logic used to communicate with the server and manage the local cache. For your users to view and modify the data fetched by LDS, you have to include UI elements. The `force:recordData` tag uses the UI API to provide data to your UI components.

When using `force:recordData`, load the data once and pass it to child components as attributes. This approach reduces the number of listeners and minimizes server calls, which improves performance and ensures that your components show consistent data.

Loading Records

The first thing you do to make a record available for your UI components is to load it. Load the record by including `force:recordData` in your component while specifying the `recordId`, `mode`, and `layoutType` or `fields` attributes.

ldsDisplayRecord.cmp

```
<aura:component implements="flexipage:availableForRecordHome, force:hasRecordId"> <!--inherit recordId attribute-->
<aura:attribute name="record" type="Object"
  description="The record object to be displayed"/>
<aura:attribute name="simpleRecord" type="Object"
  description="A simplified view record object to be displayed"/>
<aura:attribute name="recordError" type="String"
  description="An error message bound to force:recordData"/>
<force:recordData aura:id="record"
    layoutType="FULL"
    recordId="{!v.recordId}"
    targetError="{!v.recordError}"
    targetRecord="{!v.record}"
```

```
    targetFields ="{!v.simpleRecord}"
    mode="VIEW"/>
```

Copy

From here, include some markup that displays the data loaded by `force:recordData`.

```
<!-- Display a lightning card with details about the record -->
<lightning:card iconName="standard:account" title="{!v.simpleRecord.Name}" >
    <div class="slds-p-horizontal--small">
        <p class="slds-text-heading--small">
            <lightning:formattedText title="Billing City" value="{!v.simpleRecord.BillingCity}" /></p>
        <p class="slds-text-heading--small">
            <lightning:formattedText title="Billing State" value="{!v.simpleRecord.BillingState}" /></p>
    </div>
</lightning:card>
<!-- Display Lightning Data Service errors, if any -->
<aura:if isTrue="{!not(empty(v.recordError))}">
    <div class="recordError">
        {!v.recordError}
    </div>
</aura:if>
</aura:component>
```

Copy

Saving Records

The magic of LDS is when you have multiple components in a Lightning application that pull from the same record data. Some of these components simply display the record data, but other components can manipulate the data itself. This component loads the record along with a short form where the user can enter a new name for the record.

ldsSaveRecord.cmp

```
<aura:component implements="flexipage:availableForRecordHome, force:hasRecordId"> <!--inherit recordId attribute-->
<aura:attribute name="record" type="Object" />
<aura:attribute name="simpleRecord" type="Object" />
<aura:attribute name="recordError" type="String" />
<force:recordData aura:id="recordEditor"
    layoutType="FULL"
    recordId="{!v.recordId}"
    targetError="{!v.recordError}"
    targetRecord="{!v.record}"
    targetFields ="{!v.simpleRecord}"
    mode="EDIT" />
    <!-- Display a lightning card with details about the record -->
    <lightning:card iconName="standard:account" title="{!v.simpleRecord.Name}" >
        <div class="slds-p-horizontal--small">
            <p class="slds-text-heading--small">
                <lightning:formattedText title="Billing State" value="{!v.simpleRecord.BillingState}" /></p>
            <p class="slds-text-heading--small">
                 <lightning:formattedText title="Billing City" value="{!v.simpleRecord.BillingCity}" /></p>
        </div>
    </lightning:card>
    <br/>
    <!-- Display an editing form -->
    <lightning:card iconName="action:edit" title="Edit Account">
        <div class="slds-p-horizontal--small">
            <lightning:input label="Account Name" value="{!v.simpleRecord.Name}"/>
            <br/>
            <lightning:button label="Save Account" variant="brand" onclick="{!c.handleSaveRecord}" />
        </div>
    </lightning:card>
    <!-- Display Lightning Data Service errors, if any -->
    <aura:if isTrue="{!not(empty(v.recordError))}">
        <div class="recordError">
            {!v.recordError}
        </div>
    </aura:if>
</aura:component>
```

Copy

To handle this update, create a JavaScript controller that calls the `saveRecord()` method. The `saveRecord()` method takes a single callback function, `SaveRecordResult`, as its only parameter. `SaveRecordResult` includes a `state` attribute that tells you whether the save was successful, along with other information you can use to handle the result of the operation.

LdsSaveRecordController.js

```
({
    handleSaveRecord: function(component, event, helper) {
        component.find("recordEditor").saveRecord($A.getCallback(function(saveResult) {
            if (saveResult.state === "SUCCESS" || saveResult.state === "DRAFT") {
                console.log("Save completed successfully.");
            } else if (saveResult.state === "INCOMPLETE") {
                console.log("User is offline, device doesn't support drafts.");
            } else if (saveResult.state === "ERROR") {
                console.log('Problem saving record, error: ' +
                           JSON.stringify(saveResult.error));
            } else {
                console.log('Unknown problem, state: ' + saveResult.state + ', error: ' + JSON.stringify(saveResult.error));
            }
        }));}
})
```

Copy

Not too bad, right? LDS handles all the heavy lifting behind the scenes, sending the request to the server and automatically updating both records.

OK, now let's deal with the rest of the CRUD.

Creating Records

To create an empty record, leave the `recordId` attribute in `force:recordData` undefined.

ldsNewRecord.cmp

```
<aura:component implements="flexipage:availableForRecordHome, force:hasRecordId">
<aura:attribute name="newContact" type="Object"/>
<aura:attribute name="simpleNewContact" type="Object"/>
<aura:attribute name="newContactError" type="String"/>
<force:recordData aura:id="contactRecordCreator"
    layoutType="FULL"
    targetRecord="{!v.newContact}"
    targetFields ="{!v.simpleNewContact}"
    targetError="{!v.newContactError}"
    />
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
    <!-- Display the new contact form -->
    <lightning:card iconName="action:new_contact" title="Create Contact">
        <div class="slds-p-horizontal--small">
            <lightning:input aura:id="contactField" label="First Name" value="{!v.simpleNewContact.FirstName}"/>
            <lightning:input aura:id="contactField" label="Last Name" value="{!v.simpleNewContact.LastName}"/>
            <lightning:input aura:id="contactField" label="Title" value="{!v.simpleNewContact.Title}"/>
            <br/>
            <lightning:button label="Save Contact" variant="brand" onclick="{!c.handleSaveContact}"/>
        </div>
    </lightning:card>
    <!-- Display Lightning Data Service errors -->
    <aura:if isTrue="{!not(empty(v.newContactError))}">
        <div class="recordError">
            {!v.newContactError}
        </div>
    </aura:if>
</aura:component>
```

Copy

In the component controller, call the `getNewRecord()` method. Once the user creates a record, use the `saveRecord()` method shown above to save it.

Note

Now, what if you're working with something other than a text field, like a picklist, checkbox, or radio button? First, we recommend using the form-based components discussed in the previous unit because you get the field mapping out-of-the-box, along with layout, validation, CRUD changes, and error handling. With `force:recordData`, you must wire up your component to the Salesforce fields you want.

ldsNewRecordController.js

```
({
    doInit: function(component, event, helper) {
        // Prepare a new record from template
        component.find("contactRecordCreator").getNewRecord(
            "Contact", // sObject type (entityAPIName)
            null,      // recordTypeId
            false,     // skip cache?
            $A.getCallback(function() {
                var rec = component.get("v.newContact");
                var error = component.get("v.newContactError");
                if(error || (rec === null)) {
                    console.log("Error initializing record template: " + error);
                }
                else {
                    console.log("Record template initialized: " + rec.apiName);
                }
            })
        );
    },
    handleSaveContact: function(component, event, helper) {
        if(helper.validateContactForm(component)) {
            component.set("v.simpleNewContact.AccountId", component.get("v.recordId"));
            component.find("contactRecordCreator").saveRecord(function(saveResult) {
                if (saveResult.state === "SUCCESS" || saveResult.state === "DRAFT") {
                    // record is saved successfully
                    var resultsToast = $A.get("e.force:showToast");
                    resultsToast.setParams({
                        "title": "Saved",
                        "message": "The record was saved."
                    });
                    resultsToast.fire();
                } else if (saveResult.state === "INCOMPLETE") {
                    // handle the incomplete state
                    console.log("User is offline, device doesn't support drafts.");
                } else if (saveResult.state === "ERROR") {
                    // handle the error state
                    console.log('Problem saving contact, error: ' +
                                JSON.stringify(saveResult.error));
                } else {
                    console.log('Unknown problem, state: ' + saveResult.state +
                                ', error: ' + JSON.stringify(saveResult.error));
                }
            });
        }
    }
})
```

Copy

This helper validates the form values.

ldsNewRecordHelper.js

```
({
    validateContactForm: function(component) {
        var validContact = true;
         // Show error messages if required fields are blank
        var allValid = component.find('contactField').reduce(function (validFields, inputCmp) {
            inputCmp.showHelpMessageIfInvalid();
            return validFields && inputCmp.get('v.validity').valid;
        }, true);
        if (allValid) {
            // Verify we have an account to attach it to
            var account = component.get("v.newContact");
```

```
            if($A.util.isEmpty(account)) {
                validContact = false;
                console.log("Quick action context doesn't have a valid account.");
            }
        return(validContact);

        }
        }

})
```

Copy

Deleting Records

Finally, to delete a record, specify the `recordId` with the fields attribute set to “Id” at a minimum.

ldsDeleteRecord.cmp

```
<aura:component implements="flexipage:availableForRecordHome,force:hasRecordId">
<aura:attribute name="recordError" type="String" access="private"/>
<force:recordData aura:id="recordHandler"
    recordId="{!v.recordId}"
    fields="Id"
    targetError="{!v.recordError}"
    />
    <!-- Display the delete record form -->
    <lightning:card iconName="action:delete" title="Delete Record">
        <div class="slds-p-horizontal--small">
            <lightning:button label="Delete Record" variant="destructive" onclick="{!c.handleDeleteRecord}"/>
        </div>
    </lightning:card>
    <!-- Display Lightning Data Service errors, if any -->
    <aura:if isTrue="{!not(empty(v.recordError))}">
        <div class="recordError">
            {!v.recordError}
        </div>
    </aura:if>
</aura:component>
```

Copy

In the component’s JavaScript controller, call the `deleteRecord()` method. LDS deletes the record from the cache and fires a notification. The `deleteRecord()` method takes a similar callback function as the `saveRecord()` method, `deleteRecordResult`, which tells you whether the operation was successful.

ldsDeleteRecordController.js

```
({
    handleDeleteRecord: function(component, event, helper) {
    component.find("recordHandler").deleteRecord($A.getCallback(function(deleteResult) {
        if (deleteResult.state === "SUCCESS" || deleteResult.state === "DRAFT") {
            console.log("Record is deleted.");
            var resultsToast = $A.get("e.force:showToast");
            resultsToast.setParams({
                "title": "Deleted",
                "message": "The record was deleted."
            });
            resultsToast.fire();
        }
        else if (deleteResult.state === "INCOMPLETE") {
            console.log("User is offline, device doesn't support drafts.");
        }
        else if (deleteResult.state === "ERROR") {
            console.log('Problem deleting record, error: ' +
                        JSON.stringify(deleteResult.error));
        }
        else {
            console.log('Unknown problem, state: ' + deleteResult.state +
                        ', error: ' + JSON.stringify(deleteResult.error));
        }
    }));
})
```

Copy

Asynchronous Record Saving

Hypothetical situation time! So you're using the Salesforce app, and a save attempt fails to reach the server due to connection issues. Maybe the train you're on went into a tunnel, you accidentally roamed through that corner of the building without cell reception, or gremlins have been messing with the cell towers again. In any case, don't worry, LDS has your back. In the event of a connection problem, Lightning Data Service stores your changes in a local cache. This is indicated by a `DRAFT` state in the `SaveRecordResult` object. A record's `DRAFT` state is resolved when the connection is restored. When saving a record through LDS, the local cache isn't updated until the save is complete. When the save is successfully completed on the server, the cache is updated to the latest version of the record from the server, and all components with a reference to that record are notified. You don't have to worry about manually reloading the record into the cache after saving. LDS handles all the work for you.

Saves that occur when a device is offline result in a `DRAFT` state if you enable asynchronous save permissions, or if all of the following are true:

- The client can't reach the server.
- The org has enabled offline drafts.
- You have version 9.0 or newer of the Salesforce app.

All CRUD operations try to resolve immediately with an XMLHttpRequest to the server. If your device loses connection to the server, Lightning Data Service can get data from a local cache. Whether LDS pulls data from the local cache or the server depends on how old the record is, or if a local draft exists. If a record is new enough, or if a local draft exists, LDS uses the local cache. LDS only refreshes records when needed, meaning that all refreshes are triggered by a component.

Create a Trailhead Playground

Great news! You can practice using LDS in a free Trailhead Playground (TP) org. What's a TP? It's a Salesforce Developer Edition org customized for you to use with Trailhead. You can launch a TP—or create a fresh one—from any hands-on challenge. Go ahead and create a new TP now (using an existing org can create problems when you check the challenge). Scroll to the bottom of this page. Click the down arrow next to **Launch** and choose **Create a Trailhead Playground** (login required). If you ever need the login credentials for your new TP, follow the instructions in this article.

Resources

- UI API Developer Guide: Response Bodies: Record
- Lightning Data Service: Loading a Record
- Lightning Data Service: Saving a Record
- Lightning Data Service: SaveRecordResult
- Lightning Data Service: Creating a Record
- Lightning Data Service: Deleting a Record
- Lightning Data Service: Example

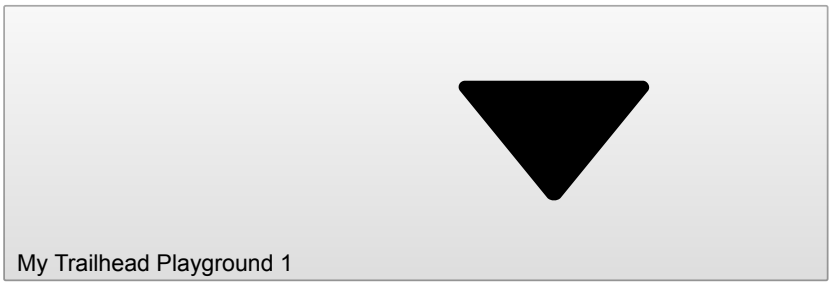
Get Ready

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

Use force:recordData to create two Aura components that display and edit the details of an account
Create Aura components to manage accounts. Use force:recordData to create a component that displays the details of a standard account record, and another component that allows for quick edits to that record.

- Create a display component named **accDisplay.cmp** using the record attribute named **accountRecord** that displays the record and record's **Name** using lightning:card, its **Industry** and **Description** using lightning:formattedText, and **Phone** using lightning:formattedPhone
- Create an edit component named **accEdit.cmp** using the record attribute named **accountRecord** that allows edits to the **Name** field. Use the **fields** attribute to query for **Name**.
- Add the following UI elements to **accEdit.cmp**:
  - lightning:input - "Account Name"
  - lightning:button - "Save Account"
- In the Salesforce UI, use the Lightning App Builder to add **accDisplay.cmp** and **accEdit.cmp** to the account record home page layout. Check out the Lightning App Builder module if you need a refresher.



Launch

Check challenge to earn 500 points