

## What You'll be Doing to Earn This Superbadge

1. Convert Salesforce Lightning Design System markup into a fully functional Lightning component.
2. Theme Lightning components using the Lightning Design System and custom CSS.
3. Surface Lightning components in Lightning App Builder, Lightning Experience, the Salesforce App, and a Lightning application.
4. Create and invoke Apex controller methods to read data from custom objects.
5. Use component events and public methods to enable communication between tightly coupled components.
6. Create and raise application events to enable communication between loosely coupled components.
7. Raise application events to invoke native Salesforce functionality.
8. Dynamically enable and disable application functionality depending on whether a feature is available in the deployment environment.
9. Use Lightning Data Service to read and write custom object data.
10. Create and use external JavaScript in a Lightning component.
11. Troubleshoot your JavaScript and CSS.

## Concepts Tested in This Superbadge

- Developing components for use in Lightning App Builder
- Theming components
- Using JavaScript to handle user interactions
- Troubleshooting components
- Dynamically showing and hiding UX controls
- Reading and writing custom object data
- Communicating between components
- Leveraging native Salesforce functionality
- Using external JavaScript in a Lightning component
-

## Pework and Notes

- Grab a pen and paper. You may want to jot down notes as you read the requirements.
- Create a new Trailhead Playground for this superbadge. Using this org for any other modules or tasks can create problems in validating the challenge. Note that your Trailhead Playground already has My Domain turned on. Don't edit the My Domain settings; you can lock yourself out of your Trailhead Playground.
- In the **Setup > Security Controls > Session Settings** section of Salesforce Classic, disable the component cache by deactivating the setting for **Enable secure and persistent browser caching to improve performance**.
- Install [this unmanaged package](#).  
(/packagingSetupUI/ipLanding.app?apvId=04tf40000011Bh4) This package contains all schema for the Apex logic needed to complete this challenge. You won't need to make any changes to the data schema. If you have trouble installing this unmanaged package, follow the steps in [Trailhead Playground Management](#).
- Sample data will automatically be added to your org after the installation of the unmanaged package is verified in Challenge 1. If you change orgs for any reason after passing the first challenge, you may execute the static method `initData()` found in `GenerateData.apxc`.
- Use the naming conventions specified in the requirements document to ensure a successful deployment.
- Review the data schema in your modified org as you read the detailed requirements below.
- When coding controller functions, use the naming convention **function foo(component, event, helper)** as opposed to **function foo(cmp,evt,hlp)**.
- When implementing events, you are required to select the appropriate event type according to best practices as well as the event's usage in the application.

## Use Case

Over the past few years, HowWeRoll Rentals, the world's largest RV rental company, has come to dominate the recreational vehicle (RV) rental marketplace. Their tagline is, "We have great service, because that's How We Roll!" Their rental fleet includes every style of camper vehicle, from palatial mobile homes to old-school, chrome Airstream campers. If you're plagued with wanderlust, they have the cure!

As the lead Salesforce developer for HowWeRoll, you have been instrumental in making the company a huge success. In order to continue to grow revenues, the company's leadership has decided to expand beyond their core RV market and enter into the recreational boating industry, as surveys have shown that a large share of RV travelers are also boat owners. Instead of making a large investment in acquiring boats of their own, HowWeRoll plans to start a boat-sharing program whereby the company acts as a leasing agent for their customers' boats. HowWeRoll is calling this new service Friends with Boats.

You've been given a week to implement a custom Lightning page, surfaced in Lightning Experience, the Salesforce App, and a Lightning application, that enables sales associates to enter information about their customers' boats. You'll also enable your team members to post comments and ratings about their experiences when they inspect each boat.

The custom search engine that you develop enables HowWeRoll's sales associates to filter the list of boats based on boat type (such as fishing boat, pleasure boat, party boat) in order to match customer requests with the boating inventory.

## Standard Objects

You'll work with the following standard objects:

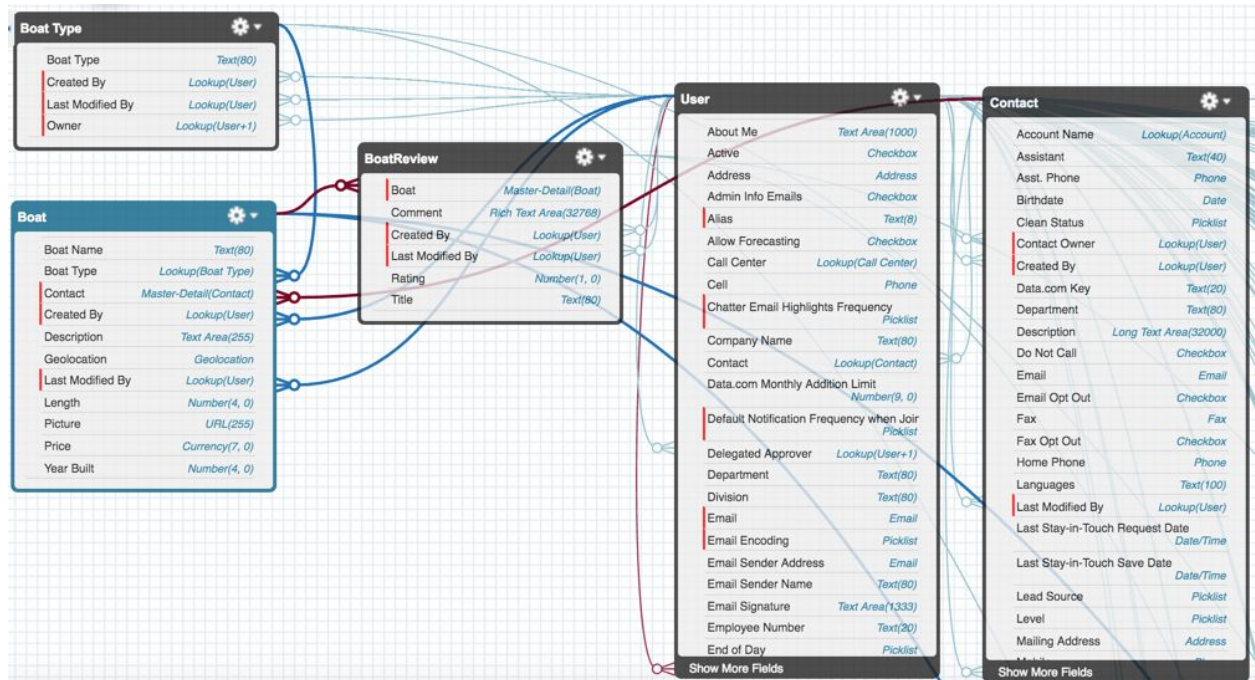
- **Contact**—Organization contacts and boat owners.
- **User**—The people posting boat reviews and comments.

## Custom Objects

- **Boat**—Information about the boats that are owned by your contacts. This object contains a geolocation field so that you can plot its typical dock location on a map. This object has a master-detail relationship with the Contact standard object and a lookup relationship with **BoatType**.
- **BoatType**—A list of the different types of boats, such as fishing boat, power boat, sailboat, party barge.
- **BoatReview**—Comments on and ratings of boats.

## Entity Diagram

From Setup, enter **Schema Builder** in the Quick Find box, then select **Schema Builder** to view an interactive version of the image. For more information, see the Work with Schema Builder unit in the [Data Modeling](#) module.



## Application Design

The time you spent meeting with key HowWeRoll stakeholders was worth your while. You came up with the following blueprint for the Lightning page.

The screenshot shows a Salesforce Lightning page titled "Friends with Boats". The page layout includes a navigation bar at the top with tabs for Sales, Opportunities, Friends with Boats (selected), Leads, Calendar, and Boat Types. Below the navigation bar, there is a search bar labeled "Find a Boat" with a dropdown menu for "All Types" and a "Search" button. To the right of the search bar is a "New" button. Below the search bar, there is a section titled "Matching Boats" displaying a grid of eight boat images, each with a name underneath: Lloyd Drucker, Jyoti Gupta, Feroz Rehman, David Gallerizzo, Keenan Keeling, Farhan Tabir, Dave Watts, and Joe Flowers. To the right of the "Matching Boats" section is a detailed view of a boat named "Lloyd Drucker's Boat". This view includes a "Full Details" button and a list of attributes: Boat Name, Type, Length, Est. Price, and Description. Below the detailed view is a map titled "Current Boat Location" showing the boat's location on a map of Washington, D.C., near the Potomac River and the Arlington National Cemetery.

After assessing the size of the task ahead, you guzzle a cup of coffee, and decide that the best way to conquer a big task is to break it into smaller pieces. You plan out the following nine phases which, when completed, will result in a solid finished product.

## Build the Search Form

A journey of a million nautical miles begins with a single line of code, right?

Get your motor running by displaying a form with a dropdown that lists each boat type, along with Search and New buttons. The form component contains the dropdown and the buttons.

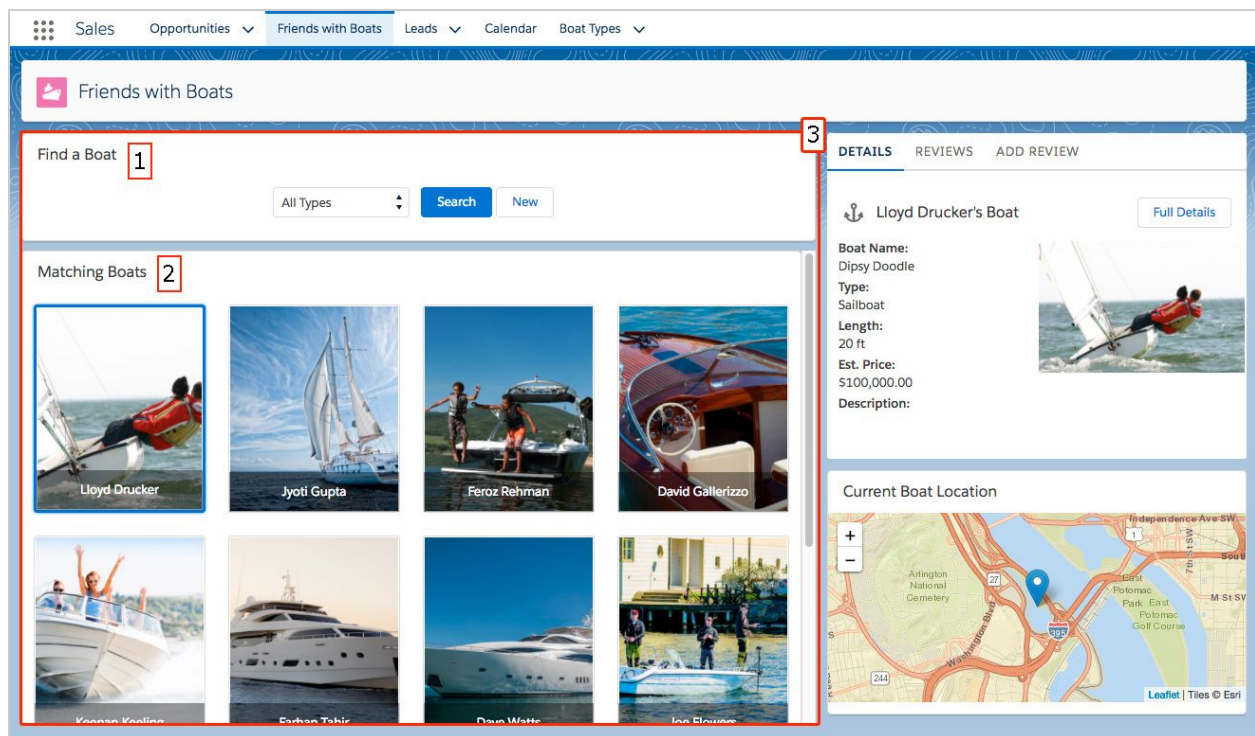


Figure 1.1: Building the Query-By-Example Interface

To build the form, create the following Lightning components. Each component is highlighted with its corresponding number in Figure 1.1.

1. **BoatSearchForm.cmp**—A form that lets users filter results by BoatType using a dropdown menu, with an empty string as the first and default value, and a label **All Types**. Include a Search button (blue **variant**) and a New button (white **variant**) that are center-aligned with the dropdown by using a `<lightning:layout>` component and its `horizontalAlign` attribute. When a user clicks **New**, a controller function fires the appropriate event to create a new Boat record. If a Boat Type is selected, the new Boat record defaults to the selected



Boat Type. The form's controller checks whether the `event.force:createRecord` event is supported by a standalone app and either shows or hides the New button according to best practices. When a user clicks **Search**, nothing happens because the filter functionality isn't implemented until the **Implement the Search Filter** phase.

2. **BoatSearchResults.cmp**—This component ultimately displays matching boats in a responsive layout, but we leave it empty for now.
3. **BoatSearch.cmp**—A container component that invokes both `BoatSearchForm.cmp` and `BoatSearchResults.cmp` and wraps each with a Lightning card with **Find a Boat** and **Matching Boats** as their respective titles. Add a margin-bottom of **10px** to the Find a Boat card to provide visual separation between the `BoatSearchForm` and `BoatSearchResults` components.

Create a Lightning page named **Friends with Boats** that uses the Main Column and Right Sidebar Layout. Put the `BoatSearch` component in the main column. Activate the page as a new tab in Lightning Experience and the Salesforce App. Lastly, create a Lightning application named `FriendswithBoats.app` that is directly accessible via its URL, with a layout that is similar to the Lightning page. Use `<lightning:layout>` to generate the app layout, and make sure that Lightning Design System styles are available to components in the app.

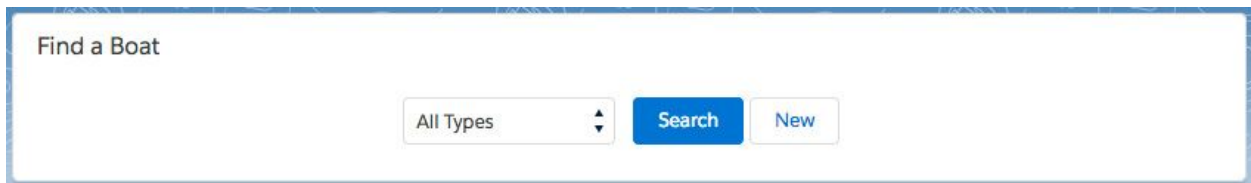
A screenshot of a Lightning card titled "Find a Boat". Inside the card, there is a search form. On the left, there is a text input field. To its right is a dropdown menu currently showing "All Types" with a small upward and downward arrow icon. Further right is a blue button labeled "Search". To the right of the "Search" button is a light blue button labeled "New".

Figure 1.2: The beginning of the search form.

Next, you begin displaying unfiltered search results in a responsive layout.

## Populate the Search Results

As fun as it is to see a list of boat types, it's time to start showing off pictures of our beautiful boats! By the end of this phase, your page will display an unfiltered list of every boat in the HowWeRoll inventory.

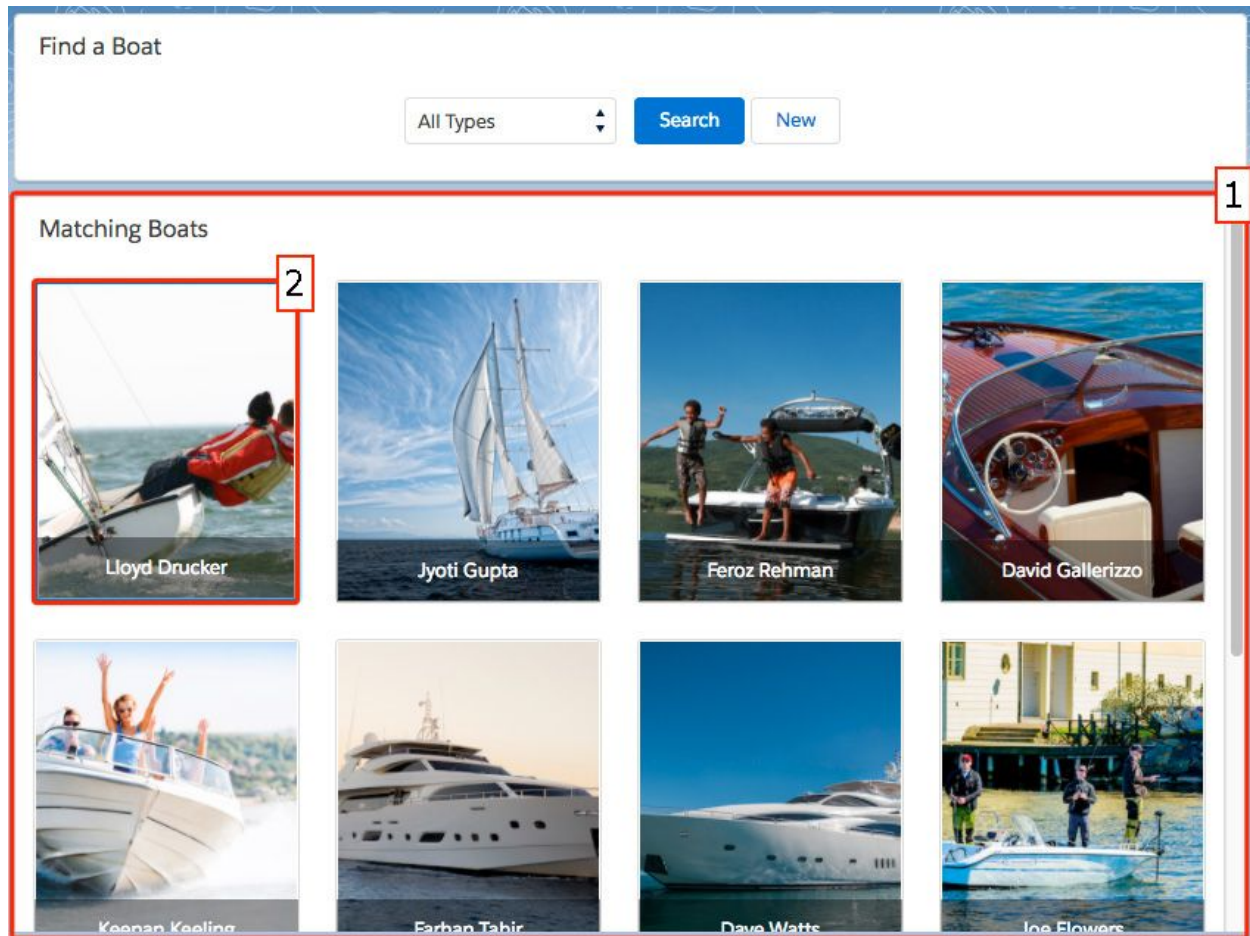


Figure 2: The **BoatSearchResults** component (1) loops through data returned by an Apex method and generates **BoatTile** (2) components.

Create a Lightning component, **BoatTile.cmp**, that displays a boat for rent and has attribute **boat** of type **Boat\_\_c**. Implement the tile as a themed **lightning:button**. Assign a **class** of **tile** to the **lightning:button** and use the following markup and CSS as a guide for how to show a boat's picture inside **BoatTile.cmp**.

### Markup

```
<lightning:button [more code here] > <div style="[set image as
background here]" class="innertile"> <div class="lower-third">
<h1 class="slds-truncate">[output contact name here]</h1> </div>
</div> </lightning:button>
```

Copy

### CSS

```
.tile { position:relative; display: inline-block; width: 100%;
height: 220px; padding: 1px !important; } .innertile {
background-size: cover; background-position: center;
background-repeat: no-repeat; width: 100%; height: 100%; }
.lower-third { position: absolute; bottom: 0; left: 0; right: 0;
color: #FFFFFF; background-color: rgba(0, 0, 0, .4); padding:
6px 8px; }
```

Copy

Use an Apex class, `BoatSearchResults`, to get the search results. This class uses a method, `getBoats()`, that takes a `boatTypeId` of type `String` and returns a list of boats filtered by that ID. When a user selects “All Types”, the empty string is passed to this function and it returns all boats.

`BoatSearchResults.cmp` gets the data returned by `BoatSearchResults` using a helper method named `onSearch()`, which stores the search results in a component attribute `boats`. Next, `BoatSearchResults.cmp` loops through the results and displays each one as a `BoatTile.cmp`, arranged in a responsive grid with multiple rows as displayed in Figure 2. Use a `<lightning:layout>` configured to allow multiple rows in order to generate the layout.

If the Apex class doesn’t return any results, display the message “No boats found” in the absolute center of `BoatSearchResults.cmp`.

In the next section, you make the search filter functional.

## Implement the Search Filter

Some people want to sail, some want to fish, and others just want to have fun. Make it easy for your associates to find the boats their customers want by hooking the search form up to the results component.



Figure 3: Implementing the search filter. Selecting a Boat Type and clicking the Search Button filters the results.

Now it’s time to modify three of your components to use an event to let the `boatTypeId` traverse the following path:

`BoatSearchForm -> BoatSearch -> BoatSearchResults -> Apex`



Use a new event `c:FormSubmit` named `formsubmit` with an `Object` attribute named `formData` to pass the selected `boatTypeId`—as a property of `formData`—from `BoatSearchForm` to its parent component `BoatSearch`. Do this via a controller function called `onFormSubmit()` attached to the Search button.

In the `BoatSearch` component, handle `FormSubmit` with a controller action named `onFormSubmit`. Pass `formData.boatTypeId` from the controller to a public method on the `BoatSearchResults` component called `search`.

Have the `search` method invoke controller function `doSearch`, which gets the `boatTypeId` parameter from the method's arguments and sets the value of the component's `boatTypeId` attribute. Then, call helper function `onSearch()`, which gets the newly updated `boatTypeId` component attribute and passes it to the server to request the list of boats. You can now display a list of boats and filter it by selecting a boat category and clicking **Search**.

## Highlight the Selected Boat

In order to get boat leases signed, the HowWeRoll sales team needs to do more than simply show some pretty pictures. They need quick access to the details for all of these boats—but first, they need to know which boat they're looking at.

Make sure it's clear to the user which boat is selected using a simple CSS rule in conjunction with a component attribute toggled by communication between components. You accomplish this by creating another event, and modifying the `BoatTile` and `BoatSearchResults` components.



Figure 4: Clicking a boat places a blue border (1) around the boat tile.

First, modify the `BoatTile` component. Add a Boolean attribute named `selected` with a default value of `false`. In the tile's `lightning:button`, use the ternary operator to add a `class` attribute that sets the class to `tile selected` if `v.selected` is true and `tile` if `v.selected` is false. Add some CSS to the component bundle by specifying the following value for the border property of the `selected` class: `3px solid rgb(0, 112, 210);`

Continue by defining a click handler on the `BoatTile`'s `lightning:button` that invokes controller function `onBoatClick`. Inside of `onBoatClick`, raise a new event called `BoatSelect`—registered on `BoatTile`—and use the event to send `boatId` to the `BoatSearchResults` component.

`BoatSearchResults` must handle the `BoatSelect` event by calling a controller function named `onBoatSelect`, which stores the `boatId` passed in via the event into a component attribute `selectedBoatId`. Use an iteration variable named `boat` and in the

`BoatSearchResults` invocation of `BoatTile`, use a ternary operator to pass a value of `true` or `false` for the `selected` attribute based on whether the currently output `boat.Id` is equal to the value stored in the component's `selectedBoatId` attribute.

## Display Boat Details

Now that it's clear which boat you're viewing, it's time to show details for the selected boat. Create a parent component `BoatDetails` with a tabset. Instantiate new child component `BoatDetail` inside of the Details tab. Use Lightning Data Service to make sure that the information is loaded properly and stays in sync with the rest of your application.



Figure 5.1: The `BoatTile` component (1) fires an event that the `BoatDetails` component (2) handles, and the display of the `BoatDetail` component (3) is updated. The `BoatDetails` component is deployed in the top right sidebar of the Lightning page as depicted in Figure 5.1. It has two public attributes: `boat` (type `Boat__c`) and `id` (type `Id`). The component outputs three tabs labeled **Details**, **Reviews**, and **Add Review**, but hides the tabset if the component's `boat` attribute is `undefined`.

Update the `BoatTile` component's `onBoatClick` handler to fire an additional new event called `BoatSelected`, passing the selected `boat` of type `Boat__c`.

The `BoatDetails` component must execute a controller function named `onBoatSelected` when the `BoatSelected` event is fired from elsewhere within the application. `onBoatSelected` sets the `id` attribute of the `BoatDetails` component to the `id` of the boat that was transmitted with the event.

The `BoatDetails` component uses the `targetFields` syntax of `force:recordData` with an `aura:id` of `service` to load the following fields from the `Boat__c` object into the `boat` attribute of the component, based upon the `BoatDetails` component's `id` attribute:

- `Id`
- `Name`
- `Description__c`
- `Price__c`
- `Length__c`
- `Contact__r.Name`
- `Contact__r.Email`
- `Contact__r.HomePhone`
- `BoatType__r.Name`
- `Picture__c`

The `force:recordData` calls an empty controller function named `onRecordUpdated` that is used in a later phase. The `BoatDetails onBoatSelected` function forces the Lightning Data Service to reload the specified record.

The `BoatDetail` component is used inside the Details tab of the `BoatDetails` component, and is invoked with an attribute `boat` of type `Boat__c`. It uses a `lightning:card` with a `utility:anchor` icon and a two-column layout built using `<lightning:layout>`. The left column displays text information about the boat, and the right column displays the boat image. The component uses the following markup and CSS to display the boat information, using the US currency format for price and allows embedded HTML in the description.

### Markup - Column 1

```
<div class="slds-p-horizontal--small"> <div
class="boatproperty"> <span class="label">Boat Name:</span>
<span></span> </div> <div class="boatproperty"> <span
class="label">Type:</span> <span></span> </div> <div
class="boatproperty"> <span class="label">Length:</span> <span>
ft</span> </div> <div class="boatproperty"> <span
class="label">Est. Price:</span> <span></span> </div> <div
class="boatproperty"> <span class="label">Description:</span>
<span></span> </div> </div>
```

Copy

### Markup - Column 2

```
<div class="imageview" style="[set image as background
here]" />
```

### CSS

```
.label { font-weight: bold; display: block; } .boatproperty {
margin-bottom: 3px; } .imageview { background-repeat: no-repeat;
background-size: contain; height: 200px; margin: 2px; }
```

Copy

The card's header is the Boat Contact's name, concatenated with "'s Boat." The Actions section of the card outputs a `lightning:button` labeled **Full Details** that calls a controller method named `onFullDetails`, which fires the appropriate event to redirect

the user to the boat's default detail page. However, the Full Details button is only output if the event is available on the deployment platform.



Figure 5.2: Create a tabset and implement the display of boat details one of the tabs. At this point, you can browse and filter boats, and view boat details. Next you add and display reviews, write secure JavaScript, and plot your boats on a map.

## Add Boat Reviews

Since HowWeRoll doesn't own all of these boats, it's important to keep track of positive and negative experiences when leasing them out to clients so that you can weed out the lemons. Accomplish this objective by adding the ability to submit reviews for each boat.

Clicking **Submit** performs the following actions:

- Creates a new record in BoatReview\_\_c (using Lightning Data Service)
- Displays a toast message that the submission was successful
- Activates the Reviews tab, which will not yet display anything

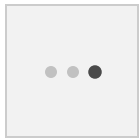


Figure 6: The Add Review form

## Build the Form

The Add Review tab of the **BoatDetails** component instantiates a new component **AddBoatReview**, passing it boat data using the component's public attribute **boat** of type Boat\_\_c. The component uses SLDS to define the form layout so that all form fields are arranged vertically. The title and description fields use appropriate Lightning components and are bound to the **Name** and **Comment\_\_c** properties of the component's private attribute **boatReview** of type BoatReview\_\_c. Font selection options are suppressed from the rich text editor for the description field. The submit button uses the **utility:save** icon and invokes a controller method, **onSave**. The rating field is not added until the **Integrate Third-Party Scripts** phase.

## Create a New BoatReview Record

The component leverages Lightning Data Service to create a BoatReview\_\_c record. Its call to **force:recordData** uses the **targetFields** syntax, has an aura:id with a value of **"service"**, references the **boatReview** attribute, and targets the following fields in the **boatReview** attribute: **Id**, **Name**, **Comment\_\_c**, **Boat\_\_c**. When the record is

updated, it invokes a controller function named `onRecordUpdated` as detailed later in this section, and it has a private component attribute named `recordError` to which it writes service errors.

On component initialization, `AddBoatReview` invokes a helper function named `onInit()` by way of controller function `doInit()`, passing along component, event, and helper arguments. The `onInit()` function invokes the appropriate method of the Lightning Data Service to get a new `BoatReview__c` record, sets the `Boat__c` property of the record to the ID of the boat that was passed into the component, and places the result into the `BoatReview` component attribute, writing any error data to the browser's JavaScript console using the `console.log()` command.

The `onSave()` controller function uses Lightning Data Service to save the record. After the record is saved, it shows a toast message if the `force:showToast` event is supported. If `force:showToast` is not supported, it displays a message using JavaScript's `alert()` method. Lastly, it invokes `helper.onInit()` to reset the component to enable the user to add another review. Similar to the `onSave()` controller function, the `onRecordUpdated()` controller function uses either a toast message or JavaScript alert to notify the user that the record has been updated.

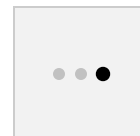
## Change the Tab Focus to the Reviews Tab

After the review has been successfully saved, a new event, `BoatReviewAdded`, is fired back to the `BoatDetails` parent component, which listens for the event, calls a controller function named `onBoatReviewAdded` and sets the currently selected tab to the Reviews tab, which has an `id` of `boatreviewtab`.

## Display Boat Reviews

Now that we've given users the ability to add reviews, let's display them.

Figure 7: Boat Reviews



### Define the Component

The Reviews tab of the `BoatDetails` component instantiates a new component, `BoatReviews`, passing it the selected boat information as a public attribute named `boat` of type `Boat__c`.

### Load the Reviews for the Selected Boat



An Apex class, **BoatReviews**, defines a function named **getAll()** that accepts an argument named **boatId** of type **Id** and returns a list of Boat Reviews containing the following fields from the BoatReview custom object:

- Id
- Name
- Comment\_\_c
- Rating\_\_c
- LastModifiedDate
- 
- CreatedDate
- CreatedBy.Name
- CreatedBy.SmallPhotoUrl
- CreatedBy.CompanyName

The **BoatReviews** Lightning component contains a private component attribute named **boatReviews** as an array of BoatReview\_\_c. The component's init handler invokes a helper function **onInit()** by way of controller function **doInit()**. The **onInit()** function communicates with Apex, placing the result into the component's **boatReviews** array attribute, handling any Apex errors gracefully.

## Output the Boat Reviews

The output of the boat reviews is based upon the [feeds component](#) of the Lightning Design System. The **BoatReviews** component defines an independently scrolling area with a max height of 250px using a Lightning component that enables native scrolling in the Salesforce app. If no reviews are found, it outputs the text "No reviews available," absolutely positioned at center within the scrollable region. If boat reviews were found, it uses an iteration variable named **boatReview** and outputs markup similar to what is described in the feeds component of the Lightning Design System, using all of the fields specified from the **BoatReview.getAll()** function as illustrated by Figure 7.

Sometimes, you want to know more about the person leaving the review, so the CreatedBy name is hyperlinked, invoking a controller function named **onUserInfoClick()**. The link contains a **data-userid** attribute that holds the value of **boatReview.CreatedBy.Id**. The **onUserInfoClick()** function retrieves the value from the **data-userid** attribute that was encoded on the hyperlink and fires an event that takes the user to the detail page of the review's author.

## Programmatically Refresh the Output

The output of the component needs to be refreshed any time a user selects a different boat or adds a new review. The component has an event handler that reloads data from

Apex any time that the value of the component's `boat` attribute is changed. It defines a public method, `refresh`, that invokes the component's `doInit()` controller method. The `refresh()` method is invoked from the `BoatDetails.onBoatReviewAdded()` method and the `BoatDetails.onRecordUpdated()` method.

## Integrate Third-Party Scripts

You're a forward thinker. When the HowWeRoll inventory grows to thousands of boats (hey, nobody ever accused you of being a pessimist!), you need to be able to see a list of your best boats at a glance. Rather than start from scratch, you begin with a script developed by an associate, because as you know—good programmers write good code, but great programmers reuse good programmers' code. The script implements a five-star rating scale. Unfortunately, your associate didn't quite finish the script so you have to fill in a few details.

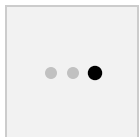


Figure 8.1: The `FiveStarRating` component in **edit** mode in the `AddBoatReview` component.

During this phase, you create the `FiveStarRating` component, which enables users to assign a rating by clicking a gold star, as illustrated by Figure 8.1.

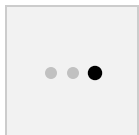


Figure 8.2: `FiveStarRating` in read-only mode in the `BoatReview` component.

The component also has a read-only mode that outputs the rating but is not clickable, as illustrated by Figure 8.2.

## Create the Component

Your newest component, `FiveStarRating`, has a `value` attribute (Integer default 0) as well as a `readonly` attribute (boolean default `false`). Load the `rating.css` and `rating.js` files from the `fivestar` static resource into it. After the script has loaded, invoke a controller function named `afterScriptsLoaded`.

Add a change event listener that invokes a controller function named `onValueChange`, which fires when the `value` attribute of the component is changed. Add a `<ul>` tag to the component that has an aura:id of `ratingarea` and uses a ternary operator to set its class attribute to either `c-rating` or `readonly c-rating` depending on the value of the `readonly` attribute.

Use the following code as the basis for your controller logic. Replace the placeholders with the appropriate code.

```
afterScriptsLoaded : function(component, event, helper) { // var
domEl = [get dom element of rating area] // var currentRating =
[get value attribute of component] // var readOnly = [get
readonly attribute of component] var maxRating = 5; var callback
= function(rating) { component.set('v.value',rating); }
component.ratingObj =
rating(domEl,currentRating,maxRating,callback,readOnly); },
onValueChange: function(component,event,helper) { if
(component.ratingObj) { var value = component.get('v.value');
component.ratingObj.setRating(value,false); } }
```

Copy

## Deploy the Component

Instantiate the component in the **AddBoatReview** component and the **BoatReviews** component. The value of the component is bound to the `Rating__c` field of the `BoatReview` custom object.

## Plot the Marker on the Map

Once they sign a lease, HowWeRoll clients need to know where to pick up the boats they're leasing. For the grand finale, you deploy a mapping component to show where your boat docks, and you also update the component to listen for a **PlotMapMarker** event from anywhere in the application.

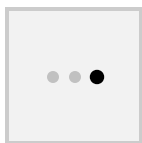


Figure 9: Clicking the **BoatTile** component (1) fires the **PlotMapMarker** event, passing latitude and longitude. The **Map** component (2) listens for **PlotMapMarker** and places a marker at the specified latitude and longitude.

The **Map** component and its controller were included in the unmanaged package that you installed as part of the prework for this superbadge, so you only need to make a few changes. Before you begin, review each of the files that came with the component to get a feel for how it works.

Create a design resource that enables a business user to set the width and height of the map component. Then, add the Map component to the Lightning page, below the **BoatDetails** component in the right sidebar as illustrated by Figure 9. Update the **Map** component by wrapping the `<div>` with `aura:id map` in a `lightning:card` with title **Current Boat Location** to keep the UI consistent with the other elements on the page. Remove the map's 1px dotted border. Then, create a new **PlotMapMarker** event which includes four string attributes: `sObjectId`, `lat`, `long`, and `label`. The event is fired when a user clicks a boat from the **BoatTile** component, but the component also listens for the event from any other component in the application. In the event listener for **PlotMapMarker**, use the latitude and longitude that were passed through the event to update the boat's location.

Complete each challenge to earn your superbadge

1

## Before you start

Complete all of the prework, including the installation of the unmanaged package.

+500 points

2

## Build the query-by-example form

Create a form displaying a dropdown that lists each boat type, along with **Search** and **New** buttons, using **BoatSearchForm.cmp**, **BoatSearchResults.cmp**, and **BoatSearch.cmp**, as described in the business requirements.

Add these components to a Lightning page named **Friends with Boats**, and activate the page as a new tab in Lightning Experience and the Salesforce App. Lastly, create a Lightning application named **FriendswithBoats.app** that has a layout that is similar to the Lightning page.

+500 points

3

## Implement the BoatTile and BoatSearchResults components

Create a new **BoatTile** component and update your **BoatSearchResults** container to loop through all the results returned from an Apex controller **BoatSearchResults** to display an unfiltered list of every boat that HowWeRoll leases.

Define the method **getBoats()** in **BoatSearchResults**, to return search results as described in the business requirements. **BoatSearchResults.cmp** displays search results with a helper method, **onSearch()**, and displays each result as a **BoatTile** component.

+500 points

4

## Implement the search filter

Create a **FormSubmit** event to allow your **BoatSearchForm** to pass the selected boat type to the **BoatSearchResults** component, which queries Apex and stores the results.

Handle **FormSubmit** with a controller action, **onFormSubmit**, and pass **formData.boatTypeId** from the controller to **search**, a public method on the **BoatSearchResults** component. The search function uses a helper function, **onSearch()**, and controller function, **doSearch()**, to get the list of boats.

+500 points

5

## Highlight the selected boat

Fire a new **BoatSelect** event when a **BoatTile** is clicked, which sets the **selectedBoatId** on **BoatSearchResults** and in turn toggles the **selected** attribute on the right **BoatTile**, triggering the addition of a CSS class that shows a dark blue border around the selected boat as shown in the requirements.

Do this by defining a click handler on the BoatTile's **lightning:button** that invokes controller function **onBoatClick**, and raises the **BoatSelect** event, as laid out in the business requirements.

+500 points



6

## Display boat details

Create two new components—**BoatDetails** and **BoatDetail**—as well as a new event **BoatSelected**.

Raise the new event from BoatTile, and leverage Lightning Data Service to output boat details. Deploy the **BoatDetails** component in the top right corner of the Lightning page.

+500 points

7

## Add boat reviews

Instantiate an **AddBoatReview** component inside the Add Review tab and display the form. When a user clicks **Submit**, save the record using Lightning Data Service and fire a **BoatReviewAdded** event that the **BoatDetails** parent component listens for so that it can switch the active tab to Reviews. Don't worry about displaying the reviews yet.

+500 points

8

## Display boat reviews

Inside the Reviews tab, invoke a **BoatReviews** component that queries Apex and outputs the results based upon the **Feeds** component of the Lightning Design System, as shown in the business requirements. Hyperlink the user's name to their record in Salesforce when possible.

+500 points

9

## Integrate third-party scripts

Create a **FiveStarRating** component with your coworker's modified JavaScript to enable associates to give the boats a rating from 1–5 stars. Give the component **edit** and **read** modes that are used in the form and output, respectively.

+500 points

10

## Plot the Map Marker on the Map

Deploy a mapping component to show where your boat docks. The **Map** component and its controller were included in the unmanaged package you installed as part of this superbadge. The component listens for the **PlotMapMarker** event, which is fired when a user clicks a boat from the **BoatTile** component.