React Week 5 Scripts

# Welcome to Week 5

Welcome to Week 5.

You will start this week by deepening your knowledge of using Redux with React. You'll learn how to split then recombine reducers, how to dispatch actions to update the state,and how to use Redux middleware.

After that, you'll learn about the fundamentals of client-server communication including the use of HTTP, and about the web services architectural style called REST.

With that knowledge, you'll learn how to make asynchronous requests to a server using the Fetch Web API, and how to handle the responses, which you'll receive as Javascript Promises.

You'll end your week by learning how to add some animation effects to your website.

There's a lot of content to get through this week, so you will have fewer Code Challenges than usual, but be sure to leave yourself enough time to complete them.

You'll also have a Quiz, and as always, you'll end the week with a workshop with your instructor and classmates.

Remember to stay focused, study every day, and don't forget the 20 minute rule.

Happy learning!

# 

# Overview - Redux Actions and Reducers

Last week, you began learning about Redux.

You took a look at how it manages the state for your React application through the use of a single store, reducers, and actions.

But what are actions exactly?

Redux Actions are plain JavaScript objects that contain payloads of information that are sent to the store.

Actions are the only way that information gets sent to the store.

The only required property for an action object is "type". The type will be a unique string.

Though it's not required, typically it's best practice to declare all the possible action types as exported string constants in a separate module. Here is an example of the ActionTypes.js file you will create this week, in its final form.

There are multiple benefits to this approach --

primarily, you are able to easily see all the action types in one place, which is very helpful when there are multiple developers on a project.

You can see in one file all the different kinds of changes to the state that have already been implemented in the app. [**screencap]**

**Dispatching an Action**

An action is dispatched by a React component, in response to some user interaction with the view.

This could be the user loading a view in the browser, clicking on a button, submitting a form, et cetera.

When Redux is used with React, you will never dispatch the action directly.

Instead, you will use the **connect()** function to do it for you, by providing it with an argument, typically called **mapDispatchToProps**. We will discuss mapDispatchToProps in more detail later.

Every time that an action is dispatched, a new action object must be created.

There is a special type of function called an action creator that is typically used to create action objects.

It's not required to use action creators to create objects, but it is helpful in making sure that objects are always created in a consistent way and to prevent typographical errors.

Here is an example of an action creator function that you will be using:

export const addCampsites = campsites => ({

type: ActionTypes.ADD\_CAMPSITES,

payload: campsites

});

This action creator will add an array of campsites to the Redux store.

It has the required type property, and it has another property called payload.

We have to call the identifier for the type property, type.

The identifier here, payload, is arbitrary, we could choose to call it something else.

We could have called it campsites, or data, for example. But the type property must be named type.

We could have other properties here aside from type and payload, as well, there isn't a limit to the number of properties you can have in an action.

For this particular action, the payload contains an array of campsites that's received as a function parameter.

Notice that because this is an arrow function, and we're only returning one thing, which is this object inside the parentheses, we don't have to use the return keyword.

Remember that with arrow functions, if you leave out the curly braces around the function body and there's only one expression in the function, you don't need to use the return keyword; it's implied.

You might be thinking, but wait, there are curly braces.

Well, curly braces do a lot of different things in JavaScript, and in this case, these curly braces are being used to create an object.

They're not the curly braces around a function body. And that's the reason that we have these parentheses around these curly braces, so we don't confuse the arrow function into thinking that these curly braces are for the function body.

So be aware that even though you don't see the return keyword here, this function is returning this object.

So, let's recap. For every action that you want to have in your app, you will be creating two things:

1, An action type defined as a string constant,

and 2, an action creator, which is a function that returns the action object.

And the action object will contain a type property, plus any data that you wish to send to the store to update it.

You also need to connect each action to your React components, and you will do that with the connect() function and a mapDispatchToProps argument.

Then every time an action is dispatched, the Redux store will check all its reducers for a matching action type, and run the code for that action type, which will then update the state.

[Slide: REDUCERS]

Now we'll discuss reducers. Here's a simplified example of a reducer you'll be implementing in a later exercise.

export const Campsites = (state = {campsites: []}, action) => {

switch (action.type) {

case ActionTypes.ADD\_CAMPSITES:

return {...state, campsites: action.payload};

default:

return state;

}

};

Reducers are considered to be part of the store. Now, while there are not multiple stores in Redux, there can be multiple reducers that handle different parts of the same state.

By the time you reach the exercise this simplified code is from, you will have done something called splitting the reducer, which creates different reducers that handle different parts of the state.

So the state that's referenced in this reducer is just the section of the entire state object that this reducer is responsible for updating.

Here this reducer takes its section of the existing state as its first argument. If that state doesn't exist yet, then it's using default function parameters here to initialize it to this object, which has the property campsites set to an empty array.

then for its second argument, it takes the action object that was dispatched.

Then it goes through this switch statement, and if any of the action types in the switch cases match the action type of the action object, it then creates a new state from the existing state, updating it with the data from the action, then returns the new state to the store.

Note that it's not required for the reducer to use a switch statement. But it's typically the best way to do it. You could use if statements or other ways to check for the action type. The only thing that's required is that the reducer return its section of the state.

Again, I will emphasize the Redux principle that State is read-only. You never mutate the existing state. You only replace it with a new state.

One of the ways that we can create the new state from the existing state is by use of the spread syntax. Here, we spread out the key-value pairs from the existing state.

Doing that doesn't mutate the existing state, we're basically making a copy of its properties Then, we add the payload from the action object into the campsites property. Then we have this all surrounded in the curly braces that signify the creation of a new object, the object literal syntax.

So remember, from the lesson on the spread syntax, since this campsites property is second in order in the object literal definition, it will overwrite the campsites property that was spread from the existing state.

Then if none of the action types matched, this default case will just return the existing state

Let's go back to a concept I mentioned briefly earlier - splitting the reducer. Last week, you just had a single file, reducer.js, which contains a single reducer function. **[open it]**

That reducer is working with the entire state, which consists of four properties, campsites, comments, partners, promotions.

These properties are all independent of each other, and can be managed separately. Often, it will be useful to split up your reducer into multiple reducers and put them in separate files.

However, Redux's createStore function only accepts a single reducer as an argument,

so Redux also provides a function called combineReducers that will take all your reducers and combine them into a single root reducer to use with createStore.

This is what you will do in the following exercise. Then after that, you will begin to implement Redux actions with action types and action creators. Let's get started.

<https://redux.js.org/basics/actions>

<https://redux.js.org/recipes/reducing-boilerplate>

<https://react-redux.js.org/using-react-redux/connect-mapdispatch#defining-mapdispatchtoprops-as-an-object>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#Advanced_syntax>

## 

# Exercise: Splitting and Combining Reducers

Added: redux/campsites.js, redux/comments.js, redux/partners.js, redux/configureStore.js

Deleted: redux/reducers.js

Changed: redux/configureStore.js

In this exercise, we will create separate reducers for each of our data arrays: campsites, comments, partners, and promotions. Then we will recombine them into a single reducer at the end.

This will replace the reducer.js file that we created previously.

[SLIDE: SPLIT THE REDUCER]

In the Redux folder, we will create 4 files: campsites.js, comments.js, partners.js, and promotions.js.

In each one, we will first import the corresponding data from the shared folder.

So for campsites.js, we'll import the CAMPSITES array from the campsites module in the shared folder.

Then we'll add a reducer function to handle each part of the state.

We will need to export it, so we'll start with that. We'll do it here as a named export. Then we will give it a name, we're choosing to capitalize it here but that isn't required.

And we're using an arrow function here but that's just a stylistic choice also, you could use a function declaration instead.

All reducers take two parameters. For the first, it takes the previous state, which can also be called the existing or current state, it just means the state that is already in the store and is going to be changed by this reducer.

The first time that a reducer is called, the state will not exist.

We'll use the default function parameter syntax to initialize the state, at least, the part of the state that's handled by this reducer, from the imported data.

For the second parameter, the reducer takes an action object.

Then in the body of this function, we will check for the type of the action, and return the state.

As I mentioned before, it's not required, but it's common to use a JavaScript switch statement for this. At this point, we do not have any action types defined, so we'll just set up a default case to return the state.

Right now, this reducer doesn't really do anything aside from initializing this part of the state if it didn't exist before.

We'll come back and add more cases to this switch statement later, and you'll see how it actually makes changes to the state then. We're just doing the setup right now.

Then we can just copy this and paste it over to each of the files we created. And make sure to change the reference to the data, so this one will be for comments.

This one will be for partners

This one will be for promotions

[SLIDE: COMBINE THE REDUCERS]

Finally, let's open configureStore.js.

As I mentioned before, the createStore function from redux will require us to give it a reducer as an argument.

But it will only accept a single reducer..

So we will need to combine our four reducers using the combineReducers function, which we need to import from 'redux'.

We can get rid of this import from the reducer module, which we are no longer using.

And we'll now need to import the separate reducers that we just created, campsites, comments, partners, promotions.

Then inside the createStore function, we can replace both the reducer and the initialstate from before with a call to combineReducers, and we'll pass it an object that contains all our reducers as properties, like this. The property identifiers here, the names for these properties, define how the data from each reducer will be kept in the overall state object tree.

At this point, you can actually delete the reducers.js file from the redux folder, as we are no longer using it. And you'll want to open your React app and make sure that it's working the same as before.

We have not made any changes to this behavior, so everything should be working just as before.

<https://redux.js.org/api/combinereducers>

# 

# Exercise: Redux Actions

* *New JavaScript: Need to explain what import \* means*

*Added: redux/ActionTypes.js, redux/ActionCreators.js*

*Changed: redux/comments.js, MainComponent.js*

In this exercise, we will finally be adding actions to our project.

We will need to create two new files in the redux folder: ActionTypes.js and ActionCreators.js.

The first one, ActionTypes.js, will be quite simple. We are only going to add one action type for now:

export const ADD\_COMMENT = 'ADD\_COMMENT';

All that this is doing is creating a variable named ADD\_COMMENT, and setting its value to the string 'ADD\_COMMENT', and exporting it.

Then in the ActionCreators.js file, we will import from the ActionTypes.js module using a wildcard.

This asterisk here acts as a wildcard that lets us import all of the named exports from the ActionTypes.js file at once. Of course. there's only one named export in ActionTypes.js right now, but we will be adding more later.

I'll show you in a moment how we can then access the exports using the ActionTypes namespace that we define here with as ActionTypes.

Next, we'll define an action creator function with the name addComment.

We'll need to pass in all the values that are needed to add a comment - the campsiteId, the rating, the author, and the text.

We'll set up this action creator to return an object which has as its properties a type and a payload.

For the type, here's where we can use that ActionTypes namespace we defined above, then dot ADD\_COMMENT. This lets us access the ADD\_COMMENT export we made from ActionTypes.js [flip back to it] without defining it explicitly here in the import.

Then for the payload property, we will pass the campsiteId, rating, author, and text.

It's worth noting that in ES6, when the identifier of a property is the same as its value, you can actually pass it like this: ..

And this will work the same way as before. That's a new feature in ES6 called shorthand property names, and I've included a link in the Additional Resources if you want to read more about it.

But I'll go ahead and return it to the way we had it before, just to make it more obvious what's happening here. You can choose to use either method.

That's it! We have just set up our first action, with an action type and an action creator. But we still have a few files to update to support this new action.

[SLIDE: Update the Reducer]

Let's go to our file for the comments reducer, comments.js. This is the one that's in the redux folder, not the one that's in the shared folder.

We will now cause this reducer to update its part of the state when the ADD\_COMMENT action is dispatched to the store. This means we need to import from the ActionTypes module, we'll do that using the wildcard syntax as you've seen before.

Then in our switch statement, we'll set up a case for when the action type is ADD\_COMMENT..

Here, we'll put the content of action.payload into a new variable named comment.

You'll recall that the content of action.payload is an object, so we can add more properties to this object like this: we'll add an id, which will be the length of the comments array that's stored in this part of the state, and we'll add today's date like this.

Then finally, we'll return the new state by using the array concat method. The concat method is built-in JavaScript array method that lets us attach a new item to the end of an array without mutating the original array - it creates a new array. In contrast, the push array method would mutate the original array, so we can't use push, we have to use concat.

So this line takes the existing state, which is an array of objects, and it concatenates the new comment object to the end of the array, then returns that new state to the Redux store.

Next, we'll update several React components to enable dispatching this action.

[SLIDE: Dispatch the Action]

Go ahead and open MainComponent.js.

First, we'll import the addComment function from ActionCreators.

And we'll now need to set up mapDispatchToProps. There's two ways of doing this - you can set up mapDispatchToProps as a function, or you can set it up as an object. The recommended way to do it is as an object. We'll set it up here as a const, and we're just going to give it one property for now -- addComment. For its value, we'll give it an arrow function with a parameter list of campsiteId, rating, author, text, and in that arrow function's body, we will call the action creator, passing in that data.

Then we need to go to the bottom of this file and add the mapDispatchToProps object inside the connect function as the second argument. Now this has made the addComment action creator function available inside the Main component as a prop.

Then in the render method of MainComponent, where we rendered the CampsiteInfo component, we can now pass the addComment function to it as a prop.

Now we need to update the CampsiteInfoComponent.js file to use that addComment action creator function. Go ahead and open that file, and scroll down to the CampsiteInfo component, which is the one that is receiving addComment as a prop from Main. From this component, we'll pass that addComment prop along to the RenderComments component, along with the campsite's id, like this.

Then let's scroll up to the RenderComments component, and we'll use object destructuring here in the parameter list to grab the props addComment and campsiteId.

Then we're going to pass these one more time to the CommentForm component, so we're not doing anything in this component with these props except just passing them to its child component, CommentForm.

Then finally in the CommentForm component, we will use these props, in the handlesubmit method. we'll add this line, this line will make it so that when the form is submitted, the addComment action creator will create an action using the values from this form. Then that action will get dispatched to its reducer, which will update the state. We can get rid of the console.log and alert here now.

Check your app in your browser. Now if you add a comment, it should appear in the comment list.

It's not stored permanently, so if you refresh the page, it will disappear. But you have now seen how to update the state using an action and a reducer.

This may seem like a lot of work right now, but remember that as an app gets more complicated, having a consistent system like this for making changes happen will make things much easier in the long run.

MDN - Import - Import an entire module's contents <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import#Import_an_entire_modules_contents>

<https://react-redux.js.org/using-react-redux/connect-mapdispatch#defining-mapdispatchtoprops-as-an-object>

# 

# Redux Middleware and Redux Thunk

In the previous lesson, we saw how we can change the state in the Redux store by dispatching actions.

At times, when an action is dispatched, we may want to intercept that action before it reaches the reducer and either change it or cause some side effect to happen, something other than changing the application state. For example, we might want to write a message to a log file. Or we might want to make a server request and wait for the server to respond.

This is where Redux Middleware can help.

Redux Middleware provides the capability to run code after an action is dispatched, and before it reaches the reducer. So middleware provides a point where you can inject third party extensions that can respond to an action. One straightforward example is Redux Logger.

Let's look at an example of how the redux-logger library works. Once you install it and add it to your React app, in the console, it will by default show a log of every single action that's dispatched. The log shows the application state prior to the action being applied, then the contents of the action object, then the new state after the reducer has applied the action. You can see how this could be very useful in debugging. You will learn how to install redux-logger in the next exercise.

Redux middleware can also help you deal with asynchronous calls. You have not dealt with asynchronous code very much at this point in your bootcamp. So let's briefly discuss what that means. With synchronous code, whenever something is executed, the application waits until it's finished before continuing. This means nothing ever happens out of order by mistake, but it can also be very slow. What if every time you tried to, for example, copy a bunch of files from one folder to another in your computer, and the whole computer froze and you couldn't do anything else until the copy operation was finished? Frustrating, right? That's what synchronous code is like.

Asynchronous code, on the other hand, will let you start something then continue on with your program without having to wait for it to finish.. You can keep going with the rest of your code, and the result of the asynchronous call will be dealt with later.

A very common asynchronous call is when your client-side code requests data from the server. This data is not available right away, the request has to go over the internet, then the server has to respond, and while typically this is very quick, there's still a small delay while that happens. So you request the data then try to use the data right away, you may encounter an error because that data isn't available yet.

We haven't accounted for this yet in the way that we've implemented the data loading in our app thus far, and it hasn't mattered because we haven't been making any server requests, we've just been grabbing the data locally. This won't work in the real world.

So how do we have an action request data from a server, then wait for and handle the server response?

So this is where Redux middleware can help us. The middleware will wrap around the action dispatch, then insert code to deal with the asynchronous operation before passing the action forward.

Aside from logging and making async calls, there are many other uses for middleware, such as crash reporting, stopping an action from reaching the reducer under certain conditions, dispatching other actions, and so on.

You can also chain middleware, so that you can run multiple third party extension libraries in sequence.

The way that you will insert Redux middleware is via the applyMiddleware function, which is passed as a parameter to the createStore function. You will see the use of this in the next lesson.

**[SLIDE: REDUX THUNK]**

Redux Thunk is one of the most useful Redux middleware libraries.

The term "thunk" itself is a programming technique where you wrap a function inside another function, allowing you to delay its execution until it's needed.

This technique comes in very handy with Redux, because it lets you inject extra operations into an action creator.

Normally, an action creator has one job - to create an action object.

Let's look at the definition of Redux Thunk from its documentation, and I want you to focus on the first couple of lines: <https://github.com/reduxjs/redux-thunk#motivation>

Redux Thunk [middleware](https://github.com/reactjs/redux/blob/master/docs/advanced/Middleware.md) allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met, or even to dispatch multiple actions.

So normally, the way that Redux is set up, the action creator function is expected to return an object which is sent directly to the reducer. Redux Thunk enables you to use that action creator to delay, stop, or change that dispatch.

This will hopefully make more sense once you're using it. In our project, what we will do is use Redux Thunk to have an action creator function generate an asynchronous request to a server for data, then dispatch a new action depending on the response to that request once it's received.

Redux Thunk is the most common way to handle the setup of simple asynchronous logic in Redux action creators. There are other ways, such as Redux Saga, which is a more advanced library suitable for handling complex asynchronous logic.

We will not go into Redux Saga in this course, but when you have mastered Redux Thunk, it's worth considering learning Saga as well. Redux-Observable is another library similar to Saga, and there are others.

But for the purposes of this course's project, Redux Thunk will be more than enough to handle our needs. In the next lesson, you will begin to use Redux Thunk and Redux Logger in your project.

<https://eloquentjavascript.net/11_async.html>

<https://medium.com/fullstack-academy/thunks-in-redux-the-basics-85e538a3fe60>

# 

# Exercise: Redux Thunk

New installs: redux-thunk, redux-logger

Added: components/LoginComponent.js

Updated: configureStore.js, campsites.js, MainComponent.js, CampsiteInfoComponent.js, DirectoryComponent.js, HomeComponent.js, ActionCreators.js, ActionTypes.js

**[Slide: Install and Enable Middleware]**

In this lesson, we will be installing both Redux Thunk and Redux Logger. Redux Logger is very simple to add, so most of this lesson will be about Redux Thunk.

First, we will need to install yarn add redux-thunk@2.2.0

yarn add redux-logger@3.0.6

Then open *configureStore.js* and update it to use Thunk and Logger as follows:

We will need to import the applyMiddleware function from the redux library  
Then we will import the redux-thunk and redux-logger libraries

And finally, in our createStore function, after our reducer, we will provide a second argument -- the applyMiddleware function

Then to enable the use of Redux thunk and logger in our app, we simply pass those values to applyMiddleware.

And for Redux Logger, that's actually all you need to do. A

Now with Redux Thunk, it is enabled for us to use, but we need to add more code in order to actually use it.

**[Add Action Types]**

Go ahead and open ActionTypes.js and let's start by adding three more action types.

This action, CAMPSITES\_LOADING, will be for when our app is loading the campsites data and it hasn't received the data yet, it's just made the request and is waiting for a response.

This action, CAMPSITES\_FAILED, will be for when our server request has failed for some reason and we weren't able to load the data, then this action will let the Redux store know that, so the state can update to show an error message.

Then this action, ADD\_CAMPSITES, is what we will dispatch when the campsites data has been successfully retrieved from the server and can be safely added to the state.

We can close that file now and let's open ActionCreators.js.

**[Add Action Creators]**

And before we edit this file, let me discuss what our objective is going to be here.

We will be using Redux Thunk to performing an asynchronous request to a server.

However, we haven't set up a server to interact with in this way yet, so for now, we're going to just pretend that we're talking to a server by simulating a brief delay using the setTimeout function.

Then after that delay, we'll go ahead and add the campsites data to the state.

We'll start by temporarily importing the campsites data into this module so we can use it in our server simulation.

Next, we'll add an action creator, and we'll call it fetchCampsites. And here is the Redux Thunk syntax - we will wrap this function in another function, and Redux Thunk lets us pass the store's dispatch method into the inner function like this. Then we can use that dispatch method here to dispatch a different action, campsitesLoading.

Then we'll use setTimeout to simulate a brief delay of 2000 milliseconds, or 2 seconds, and after that delay, we'll dispatch another action, addCampsites, along with the data from the CAMPSITES array.

So, if it's not clear to you, notice that the two arrows at the top of this function means that we've nested an arrow function inside another arrow function. It's because we enabled Redux Thunk earlier that we're able to use this syntax.

The next action creator we will write will not use Redux Thunk. So here, for the campsitesLoading action creator, you see there's only one arrow. So that means this will be your standard action creator that just returns an action object and nothing else. And this action won't have any kind of payload, we're just going to give it a type. And because it's not thunked, it's not going to be intercepted, it's just going to go straight to the reducer as normal.

And you'll notice that this action, campsitesLoading, is what was dispatched from fetchCampsites. So when fetchCampsites is dispatched, that action will dispatch this action.

Next, we'll add another action creator, this one to create the action for campsitesFailed, and this action will have a payload, an error message.

And the last action creator we'll add for now will be for the ADD\_CAMPSITES action, and this one will have the campsites array as the payload.

----

**[Slide: Update the Campsites Reducer]**

Next, we'll need to update the campsites reducer, so go ahead and open the campsites.js file that's inside the redux folder.

First, we no longer need to import the campsites data into this module. We will be receiving the campsites data from an action, so we can get rid of this import.

In its place, we'll import from our ActionTypes module.

Then here, we're going to change the structure of the campsites state. Where before, this part of the state just held the campsites array directly, now it's going to hold three different properties: a boolean isLoading property, an error message property, and then also the campsites array, and we'll initialize them all here with the default function parameters syntax.

Now we'll start adding the responses to the different campsites-related actions to this reducer's switch statement.

For the ADD\_CAMPSITES action type, we're going to return a new state that consists of the previous state, spread out, and we'll update its values to say it's no longer loading, there's no error message, and the campsites array will be populated with the payload.

Next, for the CAMPSITES\_LOADING action type, we're going to update the state to say that isLoading is true, errMess is null, and campsites is an empty array, because we haven't finished loading the data yet.

Then for CAMPSITES\_FAILED, we'll set isLoading to false, and errMess to the action's payload, and we don't need to update the campsites array for this one.

-----

Now we are done with the files in the Redux folder. We will next update our React components.

**[Update React Components]**

While we're waiting for data to be loaded from the server, what we're going to do is show a loading spinner, so let's go ahead and create a component for that view real fast.

In the components folder, create a new file named LoadingComponent.js.

Here, we'll import react, then create a functional component, and it's going to be a very simple component, all it will do is return a column div that contains a font-awesome spinner icon with some extra styling on it, along with text that says Loading.

That's all for this component. Next, go ahead and open MainCompoonent.js.

Here, we need to make the fetchCampsites action creator available. We'll start by importing it at the top.

Then we will add it to the mapDispatchToProps object, like this, so now this fetchCampsites action creator is available to the Main component as props.

And when do we want to fetch it? Well, we want to fetch it as soon as the Main component is rendered to the DOM. The best place to do this is going to be in a special React method called componentDidMount.

componentDidMount is a built-in React method that's part of a set of what's called "lifecycle methods". Every React component has what's called a lifecycle, which means, there are certain points where it gets created and inserted into the DOM, when it gets updated, and when it gets removed. These points have certain built-in methods that are called at that time. The render() method is considered a lifecycle method, for example, and a couple of other common ones are componentDidUpdate() and componentWillUnmount(). componentDidMount() is called right after a React component is created and inserted into the DOM, so that's a safe place for us to start fetching the campsites data, which is what we will do here.

Next, we need to make an adjustment to the campsites prop that we're passing into the Home component. Remember that in the campsites reducer, we restructured the part of the state that holds the campsites information? Before, this.props.campsites was holding just the campsites array. Now it's holding the isLoading and error message properties along with the campsites array, so we now need to access the campsites array like this - campsites.campsites. We're getting the campsites array out of an object that's also named campsites.

Then we'll pass the isLoading property of the campsites state object, and the error message, as props named campsitesLoading and campsitesErrMess.

Then down here, where we're passing props to the CampsiteInfo component, we'll do the same thing. We'll change this to campsites.campsites, then pass in the isLoading and errMess props.

That's all for the Main component. Next we'll update the Home component.

So here, what we want to do is show the Loading component if we're still loading data, the error message if we've somehow failed to get the data, and only render the data if we have it.

We'll start by importing the Loading component.

Then here, in the RenderCard component, we'll bring in the isLoading and errMess props.

Next we'll set up some logic here to say that if isLoading is truthy, then we'll return the Loading component. And if errMess is truthy, then we'll return the contents of that error message.

And if neither of those conditions were truthy, then we'll go ahead and return the card component that's using the campsites data, because at this point we can assume that the campsites data has loaded correctly.

Notice we could have used the if, elseif, else structure here, but I just chose not to because if either of these if conditions were truthy, then we would have returned out of this function anyway and never reached the code below them.

Finally, we need to go down to the Home component and actually pass the isLoading and errMess props to the RenderCard component, because remember, the Main component passes them to Home.

Very much like what we did in the RenderCard component, in the CampsiteInfo component, we're now going to import LoadingComponent. Then we'll add a couple if statements to check if isLoading is true, or if we have an error message, and we'll return the appropriate elements if so. And here, what's expected to be returned is a Bootstrap grid setup, so we'lll make sure to return that. And that's all for this module.

And we just have one more file to update for this exercise. Go ahead and open directorycomponent.s.

And again we'll import LoadingComponent.

Then here, where we're getting the campsites prop, we'll need to insert dot campsites here because we changed the structure of that prop.

Then down here, this should look familiar to you now, we'll set up a couple of if statements to check for if we should be displaying the loading component or an error message.

Now make sure that if you start your app in the browser, you see the Loading component any time you refresh the homepage, and the directory, and the campsiteinfo views. Also, make sure that if you open your console, you see redux-logger working.

That's the end of this exercise. And now would be a great time to make a commit if you're u sing git.

# React-Redux-Form Revisited

Added: redux/forms.js

Add new file in redux directory, forms.js, add code.

Open configureStore.js and update it to add the form to the reducers.

MainComponent.js - update

ContactComponent.js - update

At the end, form inputs should persist even if you move to a diff page and come back

and should reset once submitted and you hit OK on the alert

---

In this exercise, we will be updating the feedback form in the Contact page to use the Form component from react-redux-form instead of the Form component. This update will allow us to add more functionality to our form. First, we will be able to persist the data in the form fields, such that if we move away from the Contact page to a different view then go back to it, the form data will still be there. Also, when we submit the form, the fields will be reset.

To support this behavior, we'll first add a new file named forms.js in the redux directory. And in this file, we'll export an object that holds the initial state of the form, and we'll use this to reset the form later.

Then let's open configureStore.js. Here, we'll need to import the createForms function from react-redux-form, and we'll also import the InitialFeedback object.

Then we'll use both those imports here in the combineReducers function to set up a reducer for the feedbackForm.

Next, go ahead and open MainComponent.js. Here, we will need to import the actions object from react.redux-form. This will make an action creator named actions.reset available to us, which we'll use

here in our mapDispatchtoProps as the value for a function named resetFeedbackForm.

Then let's go all the way down to our Routes, and here we'll pass that resetFeedbackForm function to the Contact component as a prop. Since now we are passing a prop to Contact, we will change this component attribute to the render attribute, and set it up accordingly.

Finally, we'll go to the Contact component. We'll need to change this LocalForm import from react-redux-form to Form, then add the actions import as well.

Let's go to the handleSubmit method, and here we'll use that resetFeedbackForm function that we passed in, as a method of this.props.

The last thing we need to do is go to the form itself and change it from LocalForm to Form, then we'll add a model, feedbackForm, and make sure to change the closing tag for localform to form as well.

At this point, you should be able to test your React app and have the form behave in the way that I demonstrated for you in the beginning of this exercise. If you enter data into the form then move to another page and come back, the data should have persisted. And if you submit the form, after you click OK on the alert, the form should reset. Once you've confirmed that's working, go ahead and commit if you're using git, and continue to the next lesson.

<https://davidkpiano.github.io/react-redux-form/docs/api/actions.html#actions-reset>

# 

# Networking Essentials

It would be helpful at this point to go over some basics of networking , especially about how the front end communicates with the back end.

These days, the back end of web applications are typically hosted on the servers of cloud platforms such as Amazon Web Services, Heroku, DigitalOcean, Microsoft Azure, and so on.

As I've mentioned before, it's important to keep in mind that when your React app is communicating with a server, whether it's fetching data from the server or uploading data to the server, there will be delays that cannot be predicted. Aside from the normal delays associated with data moving through the network, there could also be issues -- the server may be down, or there may be an outage on the internet, there might be problems with the data, and so on.

Your front-end application needs to be able to handle such issues and keep running, and to communicate to the user about what's happening. So that's something you've already begun to do in your recent exercises, where you used Redux Thunk in your project to enable your action creator to show a Loading component, then simulate a delay before adding the campsites data.

In the following lessons,, you will be learning more about how to deal with asynchronous communication between the browser and the server. You'll learn about the meaning of a RESTful API and set up a RESTful development server. You'll learn how to request resources from a web server using the promise-based Fetch API, and more. But first, in this lesson, we'll cover a few foundational concepts.

Networking is a vast topic. There's all different kinds of networking, and actually even saying *Networking* Essentials here is a bit misleading, because we're only going to be talking about networking between web browsers and web servers specifically.

And mainly, we will discuss the way that web browsers and web servers talk to each other through a network protocol called HTTP.

First, what's a network protocol? A network protocol is a set of rules for how two networked systems are going to talk to each other. The rules define what kind of data will be exchanged, how the data will be formatted, how the systems will respond to certain types of transmissions, and so on. Without these rules, computers would not be able to communicate with each other.

There are all kinds of networking protocols, such as, TCP, UDP, FTP, SMTP, and so on. And different protocols operate at different network layers, ranging from the physical layer where you're dealing with the actual wires and circuits, to the application layer, where you're dealing more with user interface concerns.

Don't worry about these for now, though on your own time, I would suggest at least reading a little about other network protocols and layers. Right now, we will focus on HTTP, which is an application layer protocol. HTTP, or HyperText Transfer Protocol, is the foundation of data communication on the Web.

Every time you load a new webpage from the web, your browser is making a HTTP request to a server.

[DEMO]

The request doesn't go directly to the web server, there's a complicated route it takes to get there, but assuming everything is working correctly, the request will eventually arrive at the correct web server. Then the web server processes the request and sends an HTTP response back.

[show examples from MDN pngs]

The request and response will be in plain text and include

Request:

* HTTP method (such as GET, PUT, POST, DELETE)
* HTTP protocol version (typically 1.1)
* Target URL
* Header (optional)
* Body (if it exists)

Response:

* HTTP status code (such as 200, 302, 404)
* HTTP protocol version (typically 1.1)
* Header (optional)
* Body (if it exists)

An HTTP request always supplies an HTTP method. HTTP methods are sometimes called HTTP actions or HTTP verbs. It's the different types of requests that a client can make, whether the client wants to GET something, or POST something, or DELETE something. We'll discuss these in more detail later in the lesson on REST.

Both the response and request will always supply the HTTP protocol they're working with, which will typically be 1.1. There is an HTTP version 2.0 but it is not in common use yet.

The request will supply a target, typically that will be a URL, or Uniform Resource Locator, you should be familiar with what URLs look like.

Then it will typically have headers that contain more data about the request, similar to the head element in an HTML file..

Then it may or not have a body, which would contain data for the request, if needed. For example, a GET request wouldn't have a body, but a POST request would.

Then the response will always have a status code, which can range from 100 to 599. You can easily find lists of HTTP status codes and their meanings online, here's one:

<https://httpstatuses.com/>

You're probably familiar with the 404 status code which means the requested resource was Not Found. Or 403, which means Forbidden. Or 500, which means Server Not Found. So you see a pattern here, pretty much anything in the 400-500 range.

As you can see, the ones in the 100 range are informational status codes, the ones in the 200 range mean the request was successful, and the ones in the 300 range mean the request was redirected.

And the response will typically also have optional headers, and it may or may not have a body as well.

So that is the format of a HTTP request and re sponse.

[Transferring Data: XML vs JSON]

If your website is a conventional, static site like the one you built in the Bootstrap course, every time the user goes to a new page in the website, the client will make HTTP requests to the server for all the assets for that new page - the HTML, CSS, JavaScript, images, and so on.

You've learned that in a Single Page Application like you've been creating with React, it works differently. The HTML, CSS, and JavaScript files are typically downloaded at the beginning in one large download, then when the user goes to a different view, the client is only asking the server for the data for that particular view, such as the campsites data.

On the web, the two most common ways that text data is formatted are JSON and XML.

You're familiar with JSON a little bit already from using the package.json file. You can see that its format closely resembles that of JavaScript objects, and that makes sense since JSON stands for JavaScript Object Notation. It's not exactly the same, but it's very close, and that means it can easily be converted to and from JavaScript objects,

JSON is a file format that's specifically used to encode data to be transferred between the client and the server. Just like HTML is formatted in a way that's intended for browsers to handle, JSON is formatted in a way that's easy for databases and JavaScript applications to parse.

JSON is a newer file format. Before JSON, and still widely used, is a file format called XML, which looks similar to HTML but isn't meant to be read by a browser. like JSON, XML is meant to be used for storing data.

The XML format is typically used in conjunction with a communication protocol named SOAP. JSON is commonly used with RESTful web services. We will discuss both of these concepts, SOAP and especially REST, in more detail in the next lesson.

As I mentioned, JSON looks a lot like JavaScript objects, and XML looks a lot like HTML, using start and end tags. Here's an example of some data formatted in JSON, and here's that same data formatted in XML.

[Show an example of XML as well -- side by side comparison with JSON and XML]

JSON:

{

"campActivities": [

{

"activityName": "archery",

"enrollment": 7

},  
 {

"activityName": "horseback riding",

"enrollment": 12

}

}

XML:

<campActivities>

<activity>

<activityName>archery</activityName>  
 <enrollment>7</enrollment>

<activityName>horseback riding</activityName>  
 <enrollment>12</enrollment>

</activity>

</campActivities>

JSON is often preferred these days not only because it converts well to JavaScript objects, but also because it's a lot more concise - it doesn't require end tags, so there's a lot less typing.

If you are in the Full Stack bootcamp, then the last course in your bootcamp will be the Back End course, where you will learn more about setting up a real server. You won't be doing that in this course, but in an upcoming exercise, you will set up a mock server that will simulate a real server to serve up JSON data, and you'll practice fetching your data from that server instead of directly from your local files.

<https://httpstatuses.com/>

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>

## 

# Representational State Transfer (REST)

**Representational state transfer** (**REST**) is a [software architectural](https://en.wikipedia.org/wiki/Software_architecture) style that defines a set of constraints to be used for creating [Web services](https://en.wikipedia.org/wiki/Web_service). (wikipedia)

REST is short for Representational State Transfer

It is a software architectural style that defines a set of constraints for creating Web services that can communicate with clients.

Web services are basically resources that are available over the web.

There are two basic types of web services: SOAP web services, and RESTful web services.

SOAP stands for Simple Object Access Protocol. The SOAP way of building web services is the older approach. It uses called WSDL, the Web Services Description Language, for specifying how two endpoints in a network communicate with each other.

SOAP typically depends on XML for its messaging format, whereas REST is more flexible - it can use either XML, HTML, plain text, or JSON, typically JSON.

SOAP web services were the de facto standard at one point, but REST has overtaken it in popularity. The REST approach is simpler and easier to implement.

However, note that where SOAP is a protocol with strictly defined rules, REST is not. It describes an approach but does not lay out exact specifications.

<https://developer.twitter.com/en/docs>

<https://developer.wordpress.org/rest-api/>

<https://developers.google.com/maps/documentation/javascript/tutorial>

There are six defining constraints of REST. Web services that conform to these constraints are called RESTful. I'll go over each of these briefly, but be aware that this will not be an exhaustive explanation of these constraints, and you are welcome to explore the REST constraints further on your own.

1. Client/server separation. The client and server can be implemented separately from each other, as long as the interface between them stays consistent so they can still communicate. This division of UI concerns and data storage concerns improves the flexibility and scalability of both sides, and they are able to evolve independently of each other.
2. Statelessness. This means that the server does not store any information about the client's state after the communication is complete. So, when a server receives the request, the server replies, and then after that, it doesn't remember anything more about that conversation between the client and the server. Then the client is responsible for tracking its own state, whether through the front-end application state, or through storing cookies in the browser, or some other way.
3. Uniform interface. Resources that are exposed by the server must have a single URI, or Uniform Resource Identifier. This is done with a directory-like style, as you have seen with the routes you've created with react-router: <http://localhost:3000/directory/1>, for example. Also, the format
4. Cacheable. Clients can cache responses, and server responses will contain information about how to cache them, or if they should be cached at all.
5. Layered System: There can be many layers between the client and the server. The client will not necessarily know if it's dealing directly with the end server or from an intermediary server that has cached the data that's being requested.
6. Code On Demand: This is the only optional constraint. Servers may temporarily extend the functionality of client-side code by transferring executable code, such as JavaScript.

So these are the constraints that define a RESTful web service.

Furthermore, RESTful web services will be built upon the HTTP protocol and will use the HTTP methods, also known as HTTP verbs. There are 9 of them:

Slide:

POST - create a new resource on the server / CREATE

GET - request a resource from the server / READ

PUT - update a resource on the server) / UPDATE

DELETE - delete a resource on the server / DELETE

HEAD

CONNECT

OPTIONS

TRACE

CATCH

The most often used are POST, GET, PUT, and DELETE.

These methods are analogous to what's known in database programming as the CRUD operations: Create, Read, Update, and Delete. [Slide]

POST is analogous to Create, GET is analagous to READ, PUT to Update, and Delete to, well, you know. (reveal definitions in the slide list while saying this part)

Of these methods, all but POST are considered **idempotent** methods.

Idempotence is a concept in mathematics and computer science that describes when an operation can be repeated multiple times but will not change what is being operated upon aside from the first time, if at all.

For example, multiplying by 0 is an idempotent operation. If you multiply any number by 0 once, you get 0. But if you multiply it again by 0, you still have 0. And you can multiply it again by 0 a hundred more times, but you'll still have 0.

A GET request is idempotent because you can make a GET request of a resource as many times as uyou want, and it will never change the resource. GET is just reading the resource, it doesn't do anything to it.

A PUT request is idempotent because even if you accidentally send the same PUT request multiple times, it would just update the same resource with the same change.

A DELETE request is idempotent because if you request to delete a resource, and then you request to delete a resource again, it doesn't do anything more since the resource is already deleted.

A POST request however will not be idempotent because you're adding a new resource every time.

So one thing you should always keep in mind is what operations are idempotent and can be repeated without any consequences, and which ones you have to be more careful not to repeat unnecessarily.

**Resources and Representations**

A resource on a server is any piece of data that could be requested by a client. It could be a single name from a database. It could be a stock price. It could be a weather report. It could be an image. It could be an HTML file.

Then, as you saw before, the resources are made available at unique URIs, and a client can request each resource specifically by its URI. But when the server responds to that request, it does not send the resource. Instead, it sends a representation of the resource.

For example, the client might send a GET request for information about a stock from a server's database. It's stored in that particular database's format. But the server will take that information and encode it into a format that's useful to the client, which cannot read the database format.

Most of the time, that encoding will be into JSON or XML format. Then that JSON or XML data is the representation of the resource, and that representation is what actually gets send in the server response.

Or a client might send a POST request to a server, and that request would include in its body a representation of some data in JSON or XML form. The server would take that representation and use it to create a new resource. So the client and the server do not send resources to each other, they send representations which are converted to resources.

And that's why we call it Representational State Transfer.

Now that you have this basic understanding of REST and HTTP, in the next exercise, you will install an npm package for a RESTful development web server named JSON-server. And you'll continue to update your React app to communicate with that server.

## 

# Exercise: Json-Server

In this exercise, you'll set up a RESTful development server on your computer named json-server from an npm package. It will be responsible for serving up the JSON data and the image files, so that you can practice requesting those resources from your React app.

[set up data and folders]

We'll first need to set up a folder for the files that we will serve.

You can create the folder you like, whether through VS Code, or a command line, or through your File Explorer or Finder. Let's think about where to put this folder.

This folder will represent the server on the back end, which means it's not a part of your React application code, which is the front end. We will also be using json-server in the React Native class. Thus, we will create the folder for json-dash-server inside the NucampFolder directory. So we're not inside the 3-React, or nucampsite folder, we're in the NucampFolder.

Now let me go inside this new folder, then I'll create a folder inside it named public.

~~You can do this in VS Code, or in bash with the mkdir command, or just through your Windows File Explorer or MacOS Finder.~~

~~I'll do this in VS Code, so you can follow along in either OS. Go ahead and open VS Code in your 3-React folder. You can open it there directly, or if you already have VS Code open in your nucampsite folder, as I do, you can go to File -> Open Folder and choose the 3-React folder. In here, I'll make a folder named json-server. So this folder is inside the 3-React folder, NOT in the nucampsite folder.~~

~~Then in the json-server folder, create a folder inside it named public.~~

Next, we'll add some data.

Remember, in your **3-React/**nucampsite/public/assets folder, you have a folder named images. Copy that whole folder, with the images inside it, to the json-server/public folder.

There's a file you need to download from the page for this exercise: db.json. Download that file and move it to your json-server folder.

So at the end of this, your file structure should look like this:

In your ~~3-React folder~~ NucampFolder directory, you should now have a json-server folder  ~~along with the nucampsites folder.~~

inside the json-server folder, you should have the db.json file and a public folder.

in your public folder, you should have the images folder, and in that should be all the images.

Make sure that your folder and file structure matches what I have just demonstrated.

[Install and start json-server]

Now go ahead and open a bash terminal in the json-server folder. Then refer to the written instructions for the command to install json-server globally, and run that command.

~~In your bash terminal, move to your json-server folder. we'll now use npm to install json-server. Use the -g flag to install it globally. Even if you've been using yarn with your React project, it's OK to use npm here, it won't interfere with your project. If you're on a MacOS, you will most likely need to use sudo in front of it.~~

~~We could actually install this from any folder since we're installing it globally, but we'll need to be in this folder for the next step.~~

Next, we will start json-server. But first, it is very important that in Bash, we are in this json-server folder when we start the server. Because we installed json-server globally, you can now technically start json-server from any folder. So you wouldn't get an error message if you ran json-server from a different folder. But if you use the command I'm about to give you in any folder other than the json-server folder, json-server won't be able to access the data in the db.json file, or any content in the public folder. So when your React app tries to access those resources, it won't be able to. So please remember, for this bootcamp, you should only run json-server from inside this folder. If you are not 100% sure that you are in the json-server folder, type pwd, this stands for present working directory and it will show you your current location. It should look like this.

Once you have confirmed that, use this command you see here to start the server.

**json-server -H 0.0.0.0 --watch db.json -p 3001 -d 2000**

~~json-server --watch db.json -p 3001 -d 2000~~

The -H flag sets the host. In this case we're using 0.0.0.0 to make the server listen on all local IP addresses for requests, so that we can be sure that requests from the react app will go through.

This --watch flag makes the json-server keep a watch on the db.json file and reload the data any time it is modified.

This -p 3001 flag specifies that we want this server to run on port 3001. The default port number for json-server is 3000, but that's the same default port number that our react application runs on, so we have to override the default here.

Then this -d 2000 number is to simulate a delay of 2000 milliseconds, just like we were doing with the setTimeout function before, to simulate communicating with a server over the internet instead of just on our own computers.

You will want to copy this entire command down somewhere that you can find it easily, because we will be using it again in future exercises.

Then go ahead and press enter, and you should see that the json-server starts up and declares all the resources that it has available, and at what URLs. Look carefully at this screen here. Make sure that it looks just like this. It should list these resources, campsites, comments, partners, promotions, feedback. If it doesn't, that means that it wasn't able to access the db.json file that you donwloaded, so either that file is not in your json-server folder, or you did not run this command from inside the json-server folder.

In your browser, try going to this homepage for the server, and you should just see an empty set of curly braces, like this. In your terminal, you can see that json-server is showing that it received a GET request, and it couldn't find any resources to serve there. The homepage is actually pointing at the public folder, and there aren't any files there. so it's showing a HTTP status code of 404, not found. Then if you were to go to any of these other paths, you should see that particular set of data from db.json show up in your browser. And JSON-server is telling you here that this time, the GET request for /campsites was successful, so the status code is 200.

So if you're having trouble with any of your data not being loaded into your app, and you're trying to figure out why, you can check here in the terminal output from json-server to make sure it's not an issue with the server.

Let me test quickly that I can access the other resources, comments, partners, promotions, and feedback.

Then the contents of the public file are available right at the top level of this path, you don't need to type public, you can just go straight to the images folder, then if you want to access an image, just type its name. So for example, we can type: <http://localhost:3001/images/logo.png>

To shut down json-server if you ever need to, use Ctrl-C.

And one last thing I want to do with you and that's to look inside the contents of the db.json file. Let's just take a look real quick. So as you can see, inside this file, at the top level or the root level, however you want to call it,, we have what resembles a single JavaScript object literal with the curly braces, and this object has several properties, campsites, comments, partners, promotions, feedback, and all these properties are storing arrays of objects. So that's the structure of the file that we are working with. And notice that this looks a lot like JavaScript object literals, but it's not exactly the same. For example, the property identifiers here must all be inside double quotes, whereas for JavaScript objects, quotes can be used but are not required.

As I mentioned, you will be using json-server for the remaining exercises in React, and you will also be using it in your React Native course. So be sure you have followed this exercise carefully to set it up, and it works as shown, and don't forget to save the command to start json-server somewhere. And again, let me stress, during this bootcamp you must always start json-server from inside the json-server folder.

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON>

# 

# Overview - Promises

<https://javascript.info/promise-basics>

<https://developers.google.com/web/fundamentals/primers/promises>

While there's more than one way to handle asynchronous computation with JavaScript, one common way is by using promises, which were added formally to JavaScript in ES6. It was available prior to ES6 by using third party libraries.

As I've discussed before, when communicating with a server, the results are typically not available instantaneously, there's going to be a delay, even if it's very brief.

So promises are a way to let your application proceed without getting stuck waiting for that response.

OK, so what is a promise?

Well, first off, like so much in JavaScript, a promise is a special kind of object.

A promise object is a proxy for a value that is not available at the moment the promise is created.

By analogy, let's say you order a pizza. You don't get your pizza right away, but you have a promise that the pizzeria will bake and deliver the pizza to you.

Now, a promise isn't a pizza. You can't eat a promise. And a promise is not a guarantee. Anything could happen between now and when that pizza gets delivered. The pizzeria might run out of dough and not be able to make the pizza at all. The pizza delivery driver might run out of gas. A zombie apocalypse could begin.

But it's better than no promise at all. You can reasonably expect that you will either get pizza, or the pizzeria will let you know if the delivery has failed for some reason.

A promise has three states: Pending, Fulfilled, or Rejected.

Pending: You've ordered the pizza, but has not yet arrived.

Fulfilled: The pizza has been baked and delivered.

Rejected: The pizza delivery has failed for some reason.

When a promise is first created, it's in Pending state. At that point, the promise only represents a value that will be available in the future, but isn't yet.

Then when that value becomes available, then the promise will "resolve", and the promise will then be in the fulfilled state.

If for whatever reason the promise cannot be met, then the promise will "reject", and it will change to rejected state.

[Slide: CREATING A PROMISE]

To create a promise, you would do it in a similar way as when you create an object from a class. You would use the new keyword, then the word Promise capitalized, then pass a function into its constructor. And this function will have two arguments called resolve and reject. Inside that function body, you would write the code for whatever asynchronous operation it is that you want this promise to do.

Resolve and reject are callback functions, and you should call the resolve function if the asynchronous operation succeeds, passing it the value that was promised. The reject function should be called if it fails, and it should be passed an Error object. An Error object is another special type of object in JavaScript.

const somePromise = new Promise((resolve, reject) => {

/\* run some code \*/

/\* call the resolve function if successful, and the reject function if failed \*/

});

However, while it's good to know what creating a promise looks like, In your React application, you will actually not be creating any promises in this way. There's something called the Fetch API which you will be learning about, and the Fetch API will actually create promises and return them to you, and then you will handle those promises.

To handle promises, which is known as consuming a promise, you will need to know about the promise object methods called .then and .catch. There's also a method called .finally, but we will not need to use it in this course.

With the .then method, you'll give it a callback function that takes as its argument the value that's returned from the promise if it's resolved successfully. Here we're calling that value 'response'

somePromise.then(

(response) => /\* handle the response return ed from the promise's resolve function \*/ );

So that's how you handle it when the promise is resolved or fulfilled. What if it's reje cted?

There's two ways to handle that. The then method has an optional second argument, where you can write another callback function that handles the error that's returned from the promise's reject function:

somePromise.then(

(response) => /\* handle the response returned from the promise's resolve function \*/ ,

(error) => /\* handle the error returned from the reject function \*/ );

Another way is to chain a catch method at the end, and it will deal with the error if you didn't supply the second argument to the then method:

somePromise

.then((response) => /\* handle the response returned from the promise's resolve function \*/ )

.catch((error) => /\* handle the error returned from the reject function \*/ );

Sometimes you will want to have a promise generate another promise, which you can do with method chaining by having the then method return a promise, then you attach another .then method to that, then a catch at the end:

somePromise

.then((response) => /\* handle resolve response, return new promise \*/ )

.then((response) => /\* handle resolve response, return new promise \*/ )

.catch((error) => /\* handle any reject errors \*/ );

This is called a promise chain, and you can add as many promises as you like to this chain.

somePromise

.then((response) => /\* handle resolve response, return new promise \*/ )

.then((response) => /\* handle resolve response, return new promise \*/ )

.then((response) => /\* handle resolve response, return new promise \*/ )

.then((response) => /\* handle resolve response, return new promise \*/ )

.catch((error) => /\* handle any reject errors \*/ );

At the end of a promise chain, you should chain a .catch method. If any of the promises in the chain are rejected and doesn't have a reject callback, the catch method will catch that reject.

There may be times when you want to go to the catch method right away, without going through a bunch of chained methods or any other code. In that case, you can use the JavaScript statement **throw** to go immediately to the next available catch method:

somePromise

.then(

(response) => {

if (some condition) {

throw new Error("Uh oh, we have an error!"); // will go directly to the next catch

}

/\* handle resolve response, return new promise \*/

})

.then(

(response) => { /\* handle resolve response, return new promise \*/ })

.catch(

(error) => { /\* handle reject errors, or any throws \*/ });

You could have combinations of both ways to handle rejects in a promise chain, where some of the then methods have a specific way of handling errors, but there's still a catch method for those that don't.

somePromise

.then(

(response) => { /\* handle resolve response from promise, return new promise \*/ }),

(error) => { /\* handle reject response from promise /\* }

.then(

(response) => { /\* handle resolve response from promise, return new promise \*/ }),

.catch(

(error) => { /\* handle reject errors, or any throws \*/ });

Circling back to our pizza analogy, let's see how that would look with the promise syntax.

Here's a Codepen demo - you can find the link for this in the additional resources.

Down here is a function call to orderPizza.

Then in the orderPizza function, there is this promise object, pizzaPromise.

I've called a .then method on this promise, and the code inside it will run if this promise is fulfilled.

And the .catch method will run if this promise is rejected.

Then up here is the promise itself.

In here, I'm using setTimeout to simulate a delayed server response.

Then if the delivery was successful, I use the resolve function to return a pizza object.

If the delivery failed, then I use the reject function to return an error object with an error message.

Then here, you can see I have a console.log after I ordered the pizza. And I want to start a movie while I'm waiting for the pizza to be delivered. I don't want to wait until after the pizza is delivered to start the movie.

So if I run this code, in the console, you can see that I ordered the pizza, I started the movie... and then the pizza is delivered.

And if I were to change this logic up here to say that the delivery did not succeed, and run this code again, then I would get the error message instead.

<https://codepen.io/minae/pen/qBBggGO?editors=0011>

Promises are a complex topic, and there's a lot than we have the ability to cover within the scope of this course. This should have provided you with a basic understanding of what promises are and how they work, at least enough to understand what they will be doing in our React application. I would encourage you to explore the topic on your own through the additional resources. For example, in ES7, which was the update to JavaScript after ES6, a syntax called async/await was introduced that can be used instead of the then method, and that's a topic you may want to read about outside of this class.

In the next lesson, you will learn about the Fetch Web API and how you can use it now with your knowledge of promises.

<https://scotch.io/tutorials/javascript-promises-for-dummies>

<https://javascript.info/promise-basics>

<https://javascript.info/async-await>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises>

# Fetch

Remember: you can use the UP arrow key to access previous commands you've entered into bash. Use the up arrow to access previous commands, and the down arrow to go forward.

In this exercise, you'll learn about using the Fetch Web API to send requests to a server. What is a Web API? Web APIs are code that's not technically part of the JavaScript language, but are built into standard browsers by default, so that you can use them in client-side JavaScript

You've already been using many Web APIs. document.getElementById for example is part of the Document Web API. So Fetch is like that.

Before Fetch, the most common way to make client-side HTTP requests was through a Web API called XMLHttpRequest.

Fetch is meant to be its modern replacement. It provides an interface for fetching resources across networks. Fetch has a more powerful and flexible feature set, and XMLHttpRequest is now considered outdated, but you may still read about it or see it in older codebases.

There are libraries that provide alternatives to Fetch, such as Axios and Superagent, which are both built using XMLHttpRequest as the base, but wrapped with additional code that makes it more powerful. So Fetch isn't your only good option, and you may want to explore other options on your own.

[some slide]

In Fetch, you make requests to the server and get a response in return from the server. The request and response are both represented with objects. They can have a number of different properties, such as HTTP headers, the HTTP method such as get, put, post; status; credentials; body; and more.

To use fetch, you write it like a function, and it has only one required parameter, which is the path to the resource you're trying to fetch from, like this: fetch(url);

This is an example taken from some code you will be using in a future exercise. (recorded after "...this error to the catch block" below)

The Fetch API is entirely promise based. When you call fetch, its return value will be a promise containing the response object.

By default, the response will contain a boolean "ok" property. response.ok -- fetch sets this response.ok property to true or false based on the HTTP response it gets from the server, if it gets a status code like 404 for example, or anything outside of the normal ok range from 200-300, it will set it to false.

We can quickly check if a fetch was successful by checking if response.ok is true. If not, we can throw an error.

There could also be cases when you don't get a response at all, you never reached the server to even get a status code, so the promise was rejected, and we can add a second callback function here for that case, and we want to just throw this error to the catch block.

If the fetch was successful, we can chain a .then method and use a built-in method of the response object called json(). We can use this method if we know that the body will contain data in JSON format. Then it will return another promise that contains the data converted to a JavaScript object.

Then it will dispatch the addCampsites action, using that data

And if there are any issues with either the json() method, or the dispatch of addCampsites, the catch method will then dispatch the action campsitesFailed.

fetch(baseUrl + 'campsites')

.then(response => {

if (response.ok) {

return response;

} else {

const error = new Error('Error ' + response.status + ': ' + response.statusText);

error.response = response;

throw error;

}

},

error => {

const errMess = new Error(error.message);

throw errMess;

}

)

.then(response => response.json())

.then(campsites => dispatch(addCampsites(campsites)))

.catch(error => dispatch(campsitesFailed(error.message)));

This example you see here is a way of requesting data from the server, so it's a download request.

We can also upload data to the server, and you'll see how to do that as well in a later exercise.

## 

## Exercise - Fetch from Server

~~yarn add cross-fetch@2.1.0~~

Make sure json-server is running

create file named baseUrl.js in shared folder, add:

export const baseUrl = '<http://localhost:3001/>';

Update ActionTypes.js to add new action types

Update ActionCreators.js to add actions using promises for comments, partners, promotions

Update comments.js, promotions.js

Update components:

Main, Directory, Home, CampsiteInfo

We can now begin to use Fetch to communicate with the json-server.

Before you can test your app, you will need to have json-server running.

So let's get that out of the way first.

Whether using VS Code or a standalone bash terminal, open a terminal to your json-server folder. Remember, that's in your NucampFolder.

~~Then in the bash terminal, type cd ../json-server.~~

~~And assuming you set everything up correctly in the json-server exercise, you should now be in the json-server folder.~~

Now you can type in the command to start json-server again, or for a shortcut, type hit your up arrow until you see the command, then press enter.

Make sure your terminal looks like this, with the campsites, comments partners, promotions and feedback resources showing, before you continue.

~~OK, I'm going to close this terminal window here, but it will still be running in the background.~~

Open VS Code to your nucampsite project folder.

The next thing we're going to do is add a module in the shared folder of our project. Name it baseUrl.js, and here we're going to add a configuration setting that we can use in other parts of our app. We'll export this constant: export const baseUrl = '<http://localhost:3001/>'; AS you can see it's set to the url where we're running jsonsever

Now, we can use this constant wherever in our code we need to give the server address. Then if the server address ever changes, we only have to change it once in here.

Next, we will need to update several Redux files as well as React component files. Let's start with the Redux files.

[Update Redux Files]

.. update action types

.. action creators...

Here's where we're going to be adding our calls to fetch.

But first, here at the top, go ahead and change this import to an import from the baseUrl module.

Then let's go to the fetchCampsites action creator, where we've been using setTimeout to simulate a server delay.

We will now replace this timeout with a call to fetch, and we'll return the result.

We need to give fetch a URL, so that will be our baseURL for the json-server plus campsites, as that's the location for the resource we want, the campsites data.

We'll chain a then method to this. Remember, a call to fetch will return a promise. When that promise is resolved, this then method will use this json() method to convert the response from json to javascript

and that javascript will be the array of campsites.

The json() method returns a new promise for which the converted javascript array is the new response value when it resolves.

So we can chain another then method here, grab that javascript array in this campsites argument once that promise resolves, and dispatch that campsites argument with the ddCampsites action creator to be used as its payload.

We won't be dealing with errors or adding a catch method to this fetch yet. Don't worry though, we'll get to that in the next exercise.

Next, let's go to the bottom and add a new action creator, this time to fetchComments.

We'll make this a thunked action creator, with the two double arrows and the dispatch argument passed into the inner arrow function.

And we'll set up this fetch call for comments, just like we did for the campsites.

We'll set up two more action creators for handling comments, and these will be just normal action creators that return action objects, they won't be using Redux Thunk. So just one arrow function, not two. We'll set one up for comments-failed, and one for add-comments.

The next action creator will be for fetching promotions, and this one will be thunked. This action creator will look just like the one for fetchCampsites, so I'll just go and copy that over and carefully change all the references to campsites to promotions.

Then I can just copy over the loading, failed, and add action creators as well, and update all these for promotions. This saves me some typing, but when you're copying and pasting like this, you need to be extra careful to make sure you're changing everything correctly.

Now we've finished setting up our fetch calls. We will next need to update our comments and promotions reducer files. We'll start with the comments reducer.

First step, we are no longer getting the COMMENTS data from the shared folder, we're fetching it from a different server. So we can just delete this first line.

Next, just as we did with the campsites data in a previous exercise, we have now changed the structure of the data we're receiving into the comments part of the state. We're not getting a simple array of comments anymore.

We're getting an object, and it contains an error message, ErrMess, which we'll initialize here to null, and an array of comments, which we'll initialize to an empty array.

Then we can add two more cases in the switch statement:  
ADD\_COMMENTS will return the previous state, updated with a null error message, and the comments array from the action payload.

And COMMENTS\_FAILED will return the previous state, plus the error message from the action payload.

Then here, to ADD\_COMMENTS, the comment.id will no longer be state.length. That was fine when the state here was just a simple array. Now the array is stored in state.comments, so we'll insert comments here.

And again here, we can't just concat the new comment to the end anymore, we will need to spread the previous state, then update just the comments property, and that's where we'll concat the new comment.

Next we'll change the promotions reducer, and it will be very similar, except we have a loading property here.

First we'll delete this import of the promotions data

Then we'll need to add an import for the action types

Then here where we initialize the state, we'll set up an objct with isLoading initialized as true, errMess as null, and promotions as an empty array.

Then we'll add a switch case for ADD\_PROMOTIONS, which will look very similar to what we just did for adding comments, with the addition of the isLoading property

And PROMOTIONS\_LOADING, again, very similar

And PROMOTIONS\_FAILED as well.

That's all for our Redux files. Now we can move on to updating our React components.

[Update React Components]

We will need to update four React component files, starting with MainComponent.js. Don't worry, these will be small updates. The main work of this exercise has already been completed.

Go ahead and open the Main component file. We will need to import fetchComments and fetchPromotions from the action creators module.

In mapDispatchToProps, we now need to add these action creators, fetchComments ... fetch Promotions. Make sure you add a comma to the line above where you add these!

Then scroll down to the Main component itself, and here where we are passing props to the Home component, we need to insert an extra .promotions for the promotions prop, just like we did before with the campsites prop. The first promotions points to the promotions object, then the second promotions points to the promotions array inside that object.

Then we'll pass two more props - promotionLoading and promotionErrMess.

Now scroll down to where we're passing props to the CampsiteInfo component. And here we'll also insert one more "comments" into this prop. Then we'll pass the comments ErrMess prop as well.

That's all for MainComponent.js. The next file we'll update is DirectoryComponent.js.

This one will be very simple. We just need to import the baseUrl here.

Then we need to update this src attribute for the cardImg to use that baseUrl. That's it, we're all done here.

Next, go ahead and open HomeComponent. This one will need the baseUrl imported also, and we'll update this CardImg component to use the baseUrl as well. Then one more thing we need to do here, go down to the RenderCard component, and we can now pass the isLoading and errMess props.

We're almost done. Our last changes will be in CampsiteInfoComoponent, we can just copy over the baseUrl import, then again, use that in the CardImg source.

Now we are done with this exercise. Make sure to test your changes in the browser. Don't forget, json-server must be running for your app to work now . You will notice that the Node logo here is broken. Don't worry about that for now, you will be fixing that later. But everything else should be working the same, and you should see a Loading message for when the home page is loading promotions. If you open your bash terminal where json-server is running, you should be able to see the requests that the server has received, and the responses it's made. I'm not showing it here but you can also try stopping your json-server on purpose and seeing what happens to your app, just to see what happens.

Once you've confirmed that everything is working as it should, go ahead and make a commit if you're using Git. Then continue on to the next exercise, where you will be adding error handling to your fetch calls.

## 

# Exercise - Fetch Handling Errors

Update: ActionCreators.js

In our implementation of fetch thus far, we have not accounted for any errors.

In this exercise, we'll be adding error handling to our fetch Action Creators.

We'll start with fetchCampsites.

This first then method here will run when the promise that's returned from this fetch resolves, which happens when the server returns a response. But just because the server returned a response, does not mean all is well. The server could have returned a bad response, like a 404 status code. But as long as you got a response from the server, this promise considers that as being resolved, not rejected. It would be rejected if we never got a response from the server.

fetch gives us an easy way to validate the response from the server: response.ok will be true for a response with a status code in the successful range, 200-299. And it'll be false if it's not. So we'll just check if response.ok, and if so, then we return the response and it'll continue down to the next then in our promise chain. Otherwise, we'll create an error object and throw it, and it'll be caught by the catch block we'll add in a moment. status and statusText are built in properties of fetch's response object, and we'll use those here to make our error more informative.

Then, remember, a promise can resolve or reject, and we can add a second callback function as an argument to the then method to handle a rejected promise, which means we didn't get a response from the server at all, good or bad. So here we'll also create and throw an error.

Then we'll add a catch method to the end of this promise chain to catch those errors when they are thrown, and if any of the other promises in the promise chain are rejected, they will also be caught by this catch method.

Now we can just copy this over to the other fetch calls, let's do the one for fetchComments first. And we'll add a catch method at the end that will dispatch the commentsFailed action.

Then we can copy it over for fetchPromotions as well, and add a catch that will dispatch promotionsFailed.

Now to test that your error handling is working, go ahead and stop your json-server if you have it running.

THen start your app if it isn't running, and you will see Failed to Fetch messages.

If you start json-server again and refresh, those errors should go away.

That's one of the kinds of errors we set up handling for, for if the server isn't responding. Then the other kind of error can happen if the server is up and running, but we get a bad response. We can simulate that by going to ActionCreators.js and introducing a typo. Here, I'll just add an extra s here in in this fetch call. Now if we reload a view in our app that tries to fetch the campsites data, we get this other kind of error, where it gives us the response status along with the status text. I'll go ahead and change that back and make sure it's working again.

# Exercise - Fetch Post Comment

Update: ActionCreators.js, comments.js, CampsiteInfoComponent.js

[Update Redux Files]

You've seen how to GET data from a server using fetch simply by providing fetch with the URL. Now you will learn how to POST data to a server using Fetch.

We'll start by updating ActionCreator.js, and we'll go to the ADD\_COMMENT action creator. Let's actually cut this action creator and move it down below the other action creators dealing with comments, so we'll put this here, under addComments.

Then we're going to actually split this up into two different action creators. Before, we just had this one action creator to add a comment just to Redux store, which is kept locally and doesn't require any asynchronous server interaction.

Now we're going to change that, so we're going to need two different action creators. This first one, addComment, is still just going to have the job of updating the local Redux store. We're going to simplify this one quite a bit, we'll just change it so that all it does is make an action with an action type, and a payload of comment.

Then we'll make a new postComment action creator, and this one will handle the asynchronous call to fetch and actually post the new comment to the server. We'll have to make it thunked by nesting the function inside a second function as you've seen before, and we'll be using the dispatch argument, so we'll pass that into the inner function. So this action creator is now using thunk middleware so it can handle asynchronous calls inside it. And make sure to get rid of these parentheses around the return value, we no longer need them. But leave the curly braces.

Inside the curly braces, we can get rid of this action types line.

And we'll change this payload property to a constant named newComment. We'll make it an object that holds the same properties as before, the campsiteId, rating, author, and text.

One more thing we will do here is add a date to the comment object. The date isn't being passed in, we're generating a new date property right here in the action creator, and it'll just grab whatever date and time it is when this part of the code is executed.

Now we'll set up our fetch, and the first part looks like what you've seen before, we'll return a call to fetch, we'll give it a URL which is baseUrl plus comments.

Then after that, we're going to pass fetch an optional second argument, and this argument will be in the form of an object. In this object, we're going to add a property to specify the request method, POST. If you don't specify a method, the default HTTP request method for fetch is GET, which is what we've been doing until now.

Then we'll need to supply a body property for this request, and that body is going to be a JSON-encoded version of the object we created above, newComment. So that's the HTTP request body.

Next, we'll add a request header with the headers property, and the headers property needs to be an object, so it can hold one or more headers. We're just giving it one header this time, Content-Type, then application slash json, so the server knows to expect the body to be formatted as JSON.

That's it for this fetch call. So we'll close that up and now we need to handle the resolve or reject of this promise with a then method. And it's going to look a lot like what you see here in fetchPromotions, so I'm just going to copy this code here, from this then line down to this error line.

So copy that, and paste it in the postComment action creator, just like this. Take a moment to make sure your code matches what you see here.

OK, then let's finish this error line, this is what will happen if the promise is rejected.

And after that, we'll close up that then method, and chain a couple others. This will look like what you did in the other fetch calls as well. What happens when the post request is successful, is that json-server will send back the data you sent, kind of like an echo, but it will automatically insert a unique id along with it. Then you can convert that response back to javaScript, then dispatch it with the addComment action creator, so the Redux store can also be updated.

And we'll add a catch method at the end, and this will catch any rejected promises or throws and let us know that something went wrong.

We're done here with the action creators. Now we'll need to update the comments reducer.

What we need to do in the comments reducer is very simple. All we need to do is get rid of these two lines in the ADD\_COMMENT case, for the comment id and the date. We no longer need to add an id because json-server does it for us now automatically. And the date is being added in the postcomment action creator now. Next, we'll update two React component files.

[Update React Components]

In MainComponent.js, we'll go through and change every instance of addComment to postComment. I'm going to go through one by one, you could use the find and replace tool if you want, just make sure you change every instance of addComment. Starting from the top, that's going to be in this import line...

Then in mapDispatchToProps.. twice here

Then here where we're passing props to the CampsiteInfo component, twice here as well.

We can close that, and we'll do the same thing in CampsiteInfoComponent.js. We'll replace addComment with postComment in the parameter list for RenderComment..

Then inside RenderComment, where we're passing props to CommentForm, we'll change it twice

Then here in this handleSubmit method

Then twice down here, where we pass props to RenderComment.

Now you can test it in your browser. Don't forget to start your json-server before you do any testing!

When you go to the comment form and add a new comment. The new comment should persist even if you refresh the page, because it's being posted to the server, it's not just in your client state anymore.

In fact, if we took a look at the db.json file now, we can see for ourselves that the React app has updated that file. And you can see that json-server has added a unique ID to it. Make sure to test your application and make sure it's working in this way before you continue.

We've come to the end of our exercises with Fetch. Hopefully this has given you a good introduction to the world of client-server communication. I encourage you to check out the additional resources and experiment with Fetch and promises more on your own.

If you're using Git, make sure to commit after you finish this exercise.

## 

# Overview - React Animations

When I talk about animations here, I don't mean animations like a cartoon or something like that. I mean animations like fade effects, zooming and so on. Subtle animations in the user interface can improve the user experience. In this lesson and the next couple of exercises we'll take a look at how to add animations to our react application. But first, how about a preview?

Here you can see that when I load the home component, the cards will pop out slightly. Then when I navigate from one view to another, the new view will slide into place from the left, along with a subtle fade effect. And here in the CampsiteInfo component, the card does that same popout effect, and the comments fade in one at a time.

These small animations can really add visual interest to a page and keep users engaged. You might be thinking that these animations I've just shown you aren't that subtle, and you'd be right, I've exaggerated them somewhat for demonstration purposes.

There are many third party libraries to bring animations into react. We will be using two.of these,.one called React-transition-group and React-animation-components.

React-transition-group used to actually be a part of the React core library, but it was moved into a separately maintained library.

React-Transition-Group is specifically for managing transitions between views. It does so by providing a set of components that can recognize when other components are being mounted or unmounted, and help to apply transition effects during that time.

There are only four components in React-Transition-Group: Transition, CSSTransition, SwitchTransition, and TransitionGroup. We will be using two of these, CSSTransition and TransitionGroup, to create that slide effect you saw when we navigated from view to view. The CSSTransition component lets us apply CSS classes at different stages of a component's transition. So it's very open-ended and flexible, we can use whatever CSS classes we want with it. And you'll get more detail about how that works in the next exercise.The TransitionGroup component helps to manage the state of a group of components that have transition effects applied to them.

react-animation-components is the other library you will be using. And it actually makes use of the react-transition-group library itself, and builds on it. This library provides drop-in GPU-accelerated animation components. It has six components: three animation components, which are Fade, Transform, and FadeTransform. And three wrapper components, Stagger, Random, and Loop. The three that we will be using are Fade, FadeTransform, and Stagger. We won't be using the others, but you're welcome to check them out on your own.

FadeTransform lets you do both a fade and a transform of the component at once with one component. We used FadeTransform to create this effect with the card in the Home component and also in the CampsiteInfo component.

The Fade component lets you put a simple fade effect on any component or element. We are using that to fade the comments. And you see how they're coming in one at a time? That's because we're also using the Stagger component around the comments. Stagger is a wrapper component that you can use on a group to stagger a delay on a set of animation components.

In the next two exercises, you'll learn how to apply these effects.

<https://reacttraining.com/react-router/web/example/animated-transitions>

## 

# Exercise - React Animations

Install: yarn add react-transition-group@2.3.0

Update: App.css

MainComponent.js

In this exercise, we will learn to add effects when transitioning between views.

We will first need to install React-Transition-Group, ~~using yarn or npm~~.

Then we'll start by adding some CSS to App.css. It's been a while since we looked at any css, so this might look a little foreign to you.

The CSSTransition component of React-Transition-Group gives us four classes we can use: enter, enter-active, exit, and exit-active. We need to give each of these a base classname, in this case, we will use the classname "page". Then we'll create four classes, and these will help us to implement a fade effect..

.page-enter {

opacity: 0;

}

.page-enter-active {

opacity: 1;

transition: 300ms ease-out;

}

.page-exit {

opacity: 1;

}

.page-exit-active {

opacity: 0;

transition: 300ms ease-in;

}

The page-enter class and page-enter-active class describe the transition that will happen when you mount a component. It will start with opacity set to 0, then it'll transition over 300 milliseconds to an opacity of 1, using ease-out. And ease-out means that the transition will start fast, then decelerate as it reaches the end.

Then for page-exit and page-exit active, same thing but when a component is being unmounted. It'll start at opacity 1, then fade away to 0. And this transition is ease-in, which means that iti will start slow, then accelerate at the end. Even though it seems counterintuitive, it generally looks best to use ease-in for something that is going out, and ease-out for something that is coming in. Now, understand that these are CSS properties, they're not specific to the react transition group library, the library lets us use any kind of CSS we like.

We'll come back to this in a moment, but for now let's open MainComponent.js. We'll import a couple of components here from react-transition-group - CSSTransition and TransitionGroup.

Then let's go down to our routes. What we want to do here is wrap the entire Switch component in, first, a TransitionGroup component. This is just a wrapper container that helps us apply transitions to a group of components. Then inside that, we'll set up a CSSTransition component.

CSS Transition Component requires a unique key, and because the React Router assigns each route a unique key, that's available to us here as this.props.location.key.

Then we'll give it a classNames attribute, and pay attention here because this is classNames with an s on the end, not className. This is a special attribute that comes with CSSTransition.

What this does is, whatever word I give this, then it will look for a CSS rule that matches that, plus -enter, -enter-active, -exit, -exit-active, and it'll apply those CSS classes. So here in App.css, those are the classes we added. That's going to be the link between the CSS and the React component, this classNames attribute that gives the base name for the CSS classes.

And we'lll add a timeout here for the default transition period as well, too 300 milliseconds.

Now after that, let's take a look at the app. ... So once again, if you don't already have json-server running, you'll need to start it. I'll do that by navigating to the json-server folder. And then here's another way you can look up previous commands, aside from using the up arrow, which may not work too well if you've used a lot of other commands since the last time you ran json-server. You can type Ctrl+r in a bash terminal and then search for previous commands you've entered, so here I typed ctrl+r then just typed the letter j, and my command to start json-server came up.

Then I'll open a new terminal session and run yarn start there.

we take a look at the app, you can see that there's a subtle fade effect whenever you switch to a different view.

Just to demonstrate, let's add more CSS. Back in App.css, let's add a transform property to the page-enter and page-enter-active classes. translateX means we're on the horizontal plane, and we'll start at -100%, which will put the component to the left of the page. Then we'll have it move right until it reaches 0%, which is just the normal position.

So after this, you can see that whenever a component is mounted, it's moving from the left to the right.

React Transition Group

<http://reactcommunity.org/react-transition-group/>

React Router - Animated Transitions <https://reacttraining.com/react-router/web/example/animated-transitions>

MDN - CSS Transition

<https://developer.mozilla.org/en-US/docs/Web/CSS/transition>

# 

# Exercise - React Animation Components

## yarn add react-animation-components@3.0.0

## yarn add prop-types@15.6.0

Update HomeComponents.js:

In this exercise, we will add a few more animated effects to our app, using the react-animation-component library

As I mentioned before, there are many libraries that support adding animations to your React application. The ones we're using this week are far from the only ones, so you can explore others on your own if you wish.

To get started, go ahead and open your bash terminal to your project folder and yarn add ~~or npm install~~ react-animation-components. We will also need to install another library called prop-types.

Then open HomeComponent.js and let's import FadeTransform from React animation components.

This allows me to apply both fade and transform effects with the same component.

We'll add this to the Card component here

...

This *in* attribute is a Boolean that tells the component to run this transition when being mounted, and this says to transition from an initial scale of 50% to the normal value, so it's going to create kind of a pop effect, and I'll have it move a little vertically as well. You'll notice we need to use two sets of curly braces to create this object, because the first, outer set of curly braces is needed to let us embed JavaScript inside JSX.

So, you can find information about how these and other attributes for FadeTransform work in the in the documentation for react-animation-props, which is linked in the Additional Resources. And I encourage you to read through it and play around with it a little. In fact, the Code Challenge after this exercise is going to have you do just that. But be aware that documentation for these third party libraries are often not super informative, and often not really geared toward beginners. You're going to find that you can't always rely on having thorough documentation, so often you have to just try things out and figure out things, whether on your own or by asking for help. There are some projects like Bootstrap where the documentation is really great and thorough, but it's just not going to be like that all the time.

...

Let's check it out in the browser. If I refresh the homepage, I should see this popout effect with the cards when they first load. Again, not that subtle, but this is just for demonstration, you probably want to be a little more subtle on a real website. Let's see that again.

Next we'll update the CampsiteInfo component.

Again, we'll need to import react-animation-components.

Then we'll apply the same FadeTransform effect to the Card component here in RenderCampsite.

And this will be exactly the same, you could even cut and paste if you wanted.

Then for our next trick, we'll use those Stagger and Fade components we imported, down here where we render our comments.

The Stagger component works on a group of elements, like the ones we'll get from this map method.

Then we'll also apply a Fade effect to each of the comments as they're rendered.

Then it's important to remember that the unique key we set here must always go on the topmost element, so that's no longer ths div, it's the Fade component, so we need to move the key up.

After this, we can test our app in the browser once again. And when you select a campsite from the directory, you should now see that same pop out effect on the card. And you should also be able to see the comments stagger and fade in. Let's try it one more time.

Once your project is working like this, you'll be finished with this exercise. You can make a commit if you're using git, and continue on to the next lesson.

## 

# Introduction to Webpack

In the Bootstrap course, we used NPM scripts to build the distribution folder for the finished project.

We saw how to use those scripts along with third-party libraries to automate the concatenation, minification, uglification, and compression of our project files.

Create-react-app comes packaged with a utility that will help us do all those things and more, and it's called Webpack.

I won't be talking about how to configure Webpack. You won't need to do that because create-react-app configures it for you. But let's discuss for a moment what Webpack is and what it does.

Webpack is what's called a module bundler. Let's take a look at the description given on Webpack;s own website. It's a static module bundler for modern JavaScript applications. It processes your application, then builds a [dependency graph](https://webpack.js.org/concepts/dependency-graph/) which maps every module your project needs and generates one or more *bundles*.

To say that another way, it looks at the entire structure of your application, then figures out how to package your modules into the most efficient bundles. Out of the box, Webpack is only set up to work with JavaScript and JSON files. However, with the help of what are called loaders, Webpack can also process any other kind of resource or asset - HTML files, CSS, even images and other media files - Webpack loaders will transform them into a form that can be handled as if they were JavaScript modules. Along the way, Webpack will perform concatenation, minification, uglification and compression of your project files. And it will bundle them in an optimized way that can be served to the client in a single response to a request, and apply technologies such as chunk loading and prefetching. It's outside the scope of this course to go into detail about how all that works. For now, it's enough to understand that Webpack's purpose is to optimize the performance and load times of your application. In the next exercise, we'll use react-scripts to build our project's distribution files for deployment.

## 

# Exercise: Building and Deploying the React Application

The actual process of building our files is very simple, as create-react-app has already configured everything for us with webpack and a library called react-scripts. What is react-scripts? Well, before we build, let's take a peek at our package.json file.

Here, you'll notice that a dependency called react-scripts has been installed for us by create-react-app. Then when you look at the scripts down below, you'll see that react-scripts is being used for performing various operations. So when we type yarn start ~~or npm start~~, that is actually using react-scripts start. And you'll notice this build script here also actually runs react-scripts build.

We can explore a little further and go into the node-modules folder to find the react-scripts folder. Be very careful not to change anything here by accident.. If you want to know how the start script works, you can open this start.js file and look inside. If we scroll down, we'll encounter some code that tells us that the start script actually uses something called the Webpack Dev Server. So that's another nice tool that Webpack has been providing for us, the dev server that's launched whenever we run yarn ~~or npm~~ start.

Then if you look in the build.js file, you can see that Webpack is required by that file. So react-scripts is configured already to use Webpack in various ways, and we won't need to do anything with Webpack ourselves. We will just use the existing react-scripts.

Before we run the build script, it's best to clear up any warnings that come up when we yarn ~~or npm~~ start our project. So I can see there is a warning here about actions being defined but never used. We can go and fix this quickly in ContactComponent.js. If you see other warnings, try to deal with them the best you can, or if you find there's one you don't understand how to fix, please ask your instructor or ask over the Nucamp Slack channels.

Now there are no more warnings, so we'll go ahead and run the build script. And all we need to do is type ~~either~~ yarn build ~~or npm run build~~. That's it! It'll take a moment, so while we're waiting, let me just point something out to you. Remember, create-react-app automatically initializes a local git repository for your project, and it automatically generates a gitignore file as well. And if we look inside that gitignore file, we'll see that the build folder is listed in there. So that means if you do a git commit, the build folder will not be included. And that's the way you want it, because you don't need to commit the build folder, since you can always generate it with the build script when you need it.

OK, the build script is done, so now we can see that it created a build folder inside our project folder.

Let's take a look inside the build folder. Here we can see an index.html file that's been minified, then we can see the CSS files that are concatenated and minified and all that. And here you can see the JavaScript files, which have also undergone uglification and minification and all that.

In the media file, we can see the font files. And in the assets, we'll see the images, although really we don't need these anymore since we already moved the images to the server

These are all the files that are needed for our React app, and we can copy them to a web server. I won't be showing you how to put them on a real live web server on the web, but we can imitate that with the json-server we already have installed.

All we need to do is copy the contents of the build folder, then go to the json-server folder, then into the public folder there, and paste. Now json-server will serve up these files at the localhost:3001 address. Make sure you have json-server running, and if you go to that address in your browser, now you should see your React app served by json-server there.

If you want to put them on a real web server on the Web, there are many different options for that, such as Firebase, Github Pages, Amazon Web Services, Heroku, and so on. I've included some links in the Additional Resources section for this exercise that you can explore on your own. Each hosting provider will have somewhat different setup procedures.

## 

# React Week 5 Assignment

This workshop assignment will require you to use what you've learned about Redux actions, fetch, and animations.

For Task 1, you will switch from getting the partners information from the shared folder, to fetching it from json-server, as you have done with the campsites, comments, and promotions. So this will complete the transition to using Redux as the single source of truth for the application state.

You will need to set up three new action types and corresponding action creators. Let me walk you through this part of it to get you started, and also because I want to tidy up the ActionCreators file with you, which has gotten a little messy. So we'll open ActionTypes.js and group the related actions together. Then we can add the new action types for partners. So that's how you'll begin Task 1.

Then it's up to you to add the corresponding action creators for these actions. After that, you will need to set up a thunked action creator to fetch the partners data from your json-server and dispatch the other actions to the Redux store. You will need to update the partners reducer as well. As usual, this is similar to the exercises I walked you through during the week, so refer back to those exercises for clues on what to do. Finally, to complete Task 1, you will need to update three of the React component files. And in the About component, you will be asked to write a new component named PartnerList to set up the list of partners.

At the end of this task, the Node logo should have returned on the home page. The list of partners on the About page should look the same as before. But you should see a "Loading" spinner as it loads, and if json-server is not running, you should see an error message.

**Task 2**

In this task, you will use fetch to POST feedback data from the form in the Contact page so that it will persist, as you did during the week with the comment form. This feedback does not need to be displayed anywhere on the front end, so it does not need to be saved to the Redux store.

The only action you need to create for this is the post\_feedback action. When this task is completed, you should be able to submit the feedback form, then when the feedback is successfully posted, you should see an alert that looks like this. And you should be able to pull up the feedback by querying json-server like this.

**Task 3**

Apply the Stagger and Fade to the list of partners in the PartnerList component. You will find detailed instructions on this task in the written instructions.

When this task is completed, when you load the About page, the list of partners should stagger and fade in like this.

Remember to work together with your workshop partner or partners, and don't forget the 10-minute rule. Good luck everyone!

Page Text:

Objectives:

* Introduce new action types and action creators to support fetching the partners information from the server and update the Redux store
* Update the Home and About components to render the information about the partners using the fetched data from the server
* Add an animation effect to the About component where the partners information is displayed
* Enable users to submit feedback through the feedback form by creating a new feedback action creator that posts data to the json-server.

Task 1

In this task, you will switch from getting the partners information from the shared folder, to fetching it from json-server, as you have done with the campsites, comments, and promotions.

* Add new action types in ActionTypes.js to support adding, loading, and the failed loading of partners information.
* Add new action creators in ActionCreators.js to fetch partners information from the server and update the Redux store.
* Update the code in redux/partners.js to respond to the three new partners-related actions.
* Update MainComponent.js to:
  + use the new fetchPartners action creator in the appropriate places
  + update the way that the partners information is passed to the Home component
  + pass the isLoading and errMess properties of the partners object to the Home component.
* Update HomeComponent.js to:
  + Import the baseUrl constant and update the CardImg src in the RenderCard component appropriately
  + In the Home component, pass the partner's isLoading and errMess props to the RenderCard component
* Update AboutComponent.js to:
  + Create a new functional component named PartnerList that takes props as an argument. Take the declaration of **const partners** from the top of the About component and move it into this component.
  + Under this, write an if statement to handle if the partners data is loading by returning the <Loading /> component. It should only return the <Loading /> component and nothing else.
  + Under this, write an if statement to handle if there was an error message while trying to load. This should return a div with the className of "col". Inside this div should be an h4 element that contains the error message.
  + Under this, outside of any if statement, return a div with the className of "col mt-4". Inside this div, put a Media element with the "list" attribute. Inside this, embed the partners variable.
  + At the bottom of the About component, there is a div element that contains a Media list. Remove the entire div and everything inside it. In its place, render the PartnerList component, and pass it the prop "partners={props.partners}"

At the end of this task, the Node logo should have returned on the home page. The list of partners on the About page should look the same as before. But you should see a "Loading" spinner as it loads, and if json-server is not running, you should see an error message.

Task 2

In this task, you will enable posting feedback data from the form in the Contact page so that it will persist, as you did during the week with the comment form. Note: This feedback does not need to be displayed anywhere on the front end, so it does not need to be saved to the Redux store. The only action creator you will need to write for this is postFeedback.

* In ActionCreator.js:
  + Implement a new action creator named postFeedback() that takes a feedback object as an argument and posts it to the server using Fetch. This will be similar, but not identical to postComment. You will not dispatch an addFeedback action. Instead, once a response is returned and checked that it is OK, you will use a JavaScript alert() to tell the user "Thank you for your feedback", followed by the contents of the feedback object. Refer to the assignment video instructions for more details on this.
* Update MainComponent.js to:
  + Import postFeedback
  + Add postFeedback to mapDispatchToProps,
  + Pass postFeedback to the Contact component from a Route component, where you are currently only passing the resetFeedbackForm prop.
* Update ContactComponent.js to call postFeedback in the handleSubmit method, passing it the values object. This will be received in the postFeedback function in ActionCreators.js as the feedback object. Remove the alert and console.log in the handleSubmit method.
* To test that your feedback data has posted successfully, you can check it by going to <http://localhost:3001/feedback> on the json-server. (If *localhost* doesn't work, you may need to substitute 127.0.0.1) You should also be able to see the change in your db.json file directly.

Task 3

* Use the Fade and Stagger components from react-animation-components to apply animation to the list of partners in About Component.

## 

## CONCLUSION

Congratulations on completing the React course of your bootcamp! You will next continue on to React Native, where you will apply the concepts you've learned about React *and* Redux - for mobile platforms.

But before you do, let's take a quick look at what you've learned in the last 5 weeks.

We've covered a lot in 5 weeks. You started out by leveling up in your JavaScript skills - you learned about objects, classes, class inheritance, and the advanced array methods map, filter, and reduce. Through the rest of your course, you continued to add more JavaScript concepts to your arsenal, including object destructuring, computed property names, the spread syntax, how to import and export, and more.

You learned how to scaffold out a React application by using the create-react-app utility, and about the structure of a typical React application. You learned how to create different kinds of React components, how to use JSX to create React elements, and about importing and using components from third party libraries such as Reactstrap. You learned about how to set up local state in a component, and how to pass state information as props to child components, and how to use setstate to change state. You learned about the concept of Single Page Applications, and how to use the React-Router library to set up navigation within your single page app. You learned about how to get user input via controlled and uncontrolled forms.

Then later, when you learned how to use Redux for state management, you learned how to store form data in the Redux store.

Speaking of Redux, you learned first about the MVC and Flux approaches for managing application data, and how that led to the development of Redux and its approach toward state management, which hinge on the concepts of single source of truth, immutability, and one-way data flow. You learned how to use Redux actions and reducers to update the state, and Redux middleware such as Thunk to inject additional operations when actions are dispatched.

You then learned about client-server communication concepts such as HTTP, REST, JSON and XML, the Fetch API and Javascript promises. And finally, you got a taste of using third-party animation libraries, and you learned about how to build your files for deployment with Webpack, which is pre-configured with create-react-app.

The last five weeks have been a very dense download of information to your brain. You should be impressed by how much you have learned! If all the concepts don't quite make sense to you yet, that's natural. If you're feeling overwhelmed or lost at this point, don't be discouraged. It will all take time and practice to sink in. Enjoy your one week break, and we'll see you again in the React Native course.