

WEEK 15

Front-End State Management

Redux Learning Objectives: Part 1

- **Redux Explained**
 - [Where did Redux come from?](#)
 - [When is it appropriate to use Redux?](#)
 - [Vocabulary](#)
- **Flux and Redux**
 - [What is Flux?](#)
 - [Redux](#)
- **Store**
 - [Creating the store](#)
 - [Store API](#)
 - [Updating the store](#)
 - [Subscribing to the store](#)
 - [Reviewing a simple example](#)
- **Reducers**
 - [Updating the reducer to handle additional action types](#)
 - [Avoiding state mutations](#)
- **Actions**
 - [Using action creators](#)
 - [Preventing typos in action type string literals](#)
 - [Reviewing a completed Fruit Stand example](#)
- **Debugging Arrow Functions**
 - [Understanding the limitations of implicit return values](#)

- [Rewriting an arrow function to use an explicit return statement](#)
- [Redux To-do List Project](#)

Redux Learning Objectives: Part 2

- **Using Redux with React**
 - [Integrating Redux into a React application](#)
 - [Organizing your Redux code](#)
 - [Writing Redux aware React components](#)
 - [Reviewing a completed Fruit Stand example](#)
- **Splitting and Combining Reducers**
 - [Splitting reducers](#)
 - [Combining reducers](#)
 - [Delegating to reducers](#)
 - [Destructuring State in your component](#)
 - [Reviewing a completed Fruit Stand example](#)
- **Container Components**
 - [Comparing presentational and container components](#)
 - [Determining where to create containers](#)
 - [Writing a container component](#)
 - [Reviewing a completed Fruit Stand example](#)
- **Freezing Objects**
 - [Using `Object.freeze` to prevent state mutations](#)
- **Preloaded State**
 - [Creating a store with preloaded state](#)
 - [Creating local storage helper functions](#)
 - [Saving state to local storage](#)
 - [Loading state from local storage](#)
- [React and Redux To-Do List Project](#)

Redux Learning Objectives: Part 3

- **Higher-Order Components**
 - [Reviewing higher-order functions](#)

- Leveraging higher-order components (HOCs)
- Using HOCs to keep code DRY
- Reviewing a completed Fruit Stand example
- **React-Redux:** `<Provider/>`
 - Understanding the advantages of the `<Provider />` component
 - Adding `<Provider />`
 - Understanding how `<Provider />` relates to the React Context API
- **React-Redux:** `connect()`
 - Calling `connect`
 - Defining `mapStateToProps(state, [ownProps])`
 - Defining `mapDispatchToProps`
 - Putting it all together
 - Reviewing a completed Fruit Stand example
- **Selectors**
 - Writing a selector
 - Selector examples
- **Middleware**
 - Applying middleware to a Redux store
 - Reviewing the signature of middleware functions
 - Creating your own `logger` middleware
 - Installing and applying the `redux-logger` middleware
- **Thunks**
 - Looking at how thunks work
 - Reviewing a concrete example
 - Reviewing a completed Fruit Stand example
- **Advanced Containers**
 - Knowing when to break the rules
- **Redux Developer Tools**
 - Instructions
 - Use
 - Resources
- *Redux-Based Pokedex Project*

- *Giphy Search Project*

Hooks Objectives

- **Intro to React Hooks**
 - `useState`
 - `useEffect`
 - `useContext`
- **Introduction to Hooks in Redux**
 - Using hooks with Redux
 - Getting Started
 - `useSelector`
 - `useDispatch`
 - Refactoring an existing component
 - Additional resources
 - Starting point for IP address project
- **React Router Hooks**
 - `useParams`
 - `useHistory`
 - `useLocation`
 - `useRouteMatch`
 - Bring it together
- **React Hooks Documentation**
 - Why hooks?
 - Hook basics
 - More about hooks
- *Pokedex Hooks Project: Phase 1*
- *Pokedex Hooks Project: Phase 2*
- *Pokedex Hooks Project: Phase 3*

WebSockets Learning Objectives

- **WebSockets Overview**
 - Key points about Web sockets

- [Client-side code](#)
- [Server-side code](#)
- **WebSocket Server Applications**
- **WebSocket Client Applications**
- [Tic-Tac-Toe Online Project](#)

React Solo Project Expectations

- [React Project](#)
 - [Homework](#)
 - [Planning Day](#)
 - [Project ideas](#)

WEEK-15 DAY-1

Redux

Redux Learning Objectives: Part 1

You've been using React's Context API to manage global state to share the same information across multiple components. Redux, like Context, gives you a way to store and manage global state in your React applications. Even though Context has become a popular option since its introduction, Redux remains a popular option for projects with sophisticated global state requirements.

After reading and practicing how to use Redux, you should be able to:

- Describe the Redux data cycle
- Describe the role of the store in the Redux architecture
- Explain what a *reducer* is
- Use the `createStore` method to create an instance of a Redux store
- Use the `store.dispatch` method to dispatch an action to trigger a state update
- Use the `store.subscribe` method to listen for state updates
- Use the `store.getState` method to get the current state
- Use a `switch` statement within a reducer function to handle multiple action types
- Describe why it's important for a reducer to avoid mutating the current state when creating the next state
- Write an action creator function to facilitate in the creation of action objects
- Use constants to define action types to prevent simple typos in action type string literals

Redux Explained

Redux is a JavaScript framework for managing the frontend state of a web application. It allows us to store information in an organized manner in a web app and to quickly retrieve that information from anywhere in the app. Redux is modeled on a few previous web technologies including [Elm](#) and [Flux](#).

Advantages of Redux include:

- It simplifies some of the more cumbersome aspects of Flux
- It is very lightweight; the library only takes up 2 kbs
- It is very fast (the time to insert or retrieve data is low)

- It is predictable (interacting with the data store in the same way repeatedly will produce the same effect)

Where did Redux come from?

Redux was created by Dan Abramov in 2015. It was initially intended as an experiment to create a simplified version of Flux. Abramov wanted to remove some of what he saw as the unnecessary boilerplate code that was required to create a Flux app.

Abramov also wanted to eliminate some of the aspects of development he found frustrating. When trying to debug a web app, one must often go through the series of steps that cause the bug to occur each time the code is changed. This quickly becomes repetitive and frustrating. Abramov envisioned dev tools that would allow one to undo or replay a series of actions at the click of a button. This idea became the Redux DevTools.

The reason this works is that Redux updates the state using pure functions called reducers (see below for definitions), so one can simply replay a series of actions and be guaranteed to arrive at the same final state. As Redux was developed it also became more convenient to use a single object to store the state, as opposed to traditional Flux which uses multiple stores.

These design choices allowed for the creation of an ecosystem of powerful Redux tools and extensions. Over time three principles were recognized as central to the philosophy of Redux. They are:

- **A Single Source of Truth** The state for an entire Redux app is stored in a single, plain JavaScript object.
- **State is Read Only** The state object cannot be directly modified. Instead it is modified by dispatching actions.
- **Changes Are Made with Pure Functions** The reducers that receive the actions and return updated state are pure functions of the old state and the action.

Beyond this, a guiding meta-philosophy of Redux is the idea that in a software library restrictions can be just as important as features. Redux deliberately places significant restrictions on the way state can be stored and updated, but in return it allows easy implementation of a number of powerful features that would be extremely difficult to write using a less restrictive framework.

When is it appropriate to use Redux?

Initially, Redux grew in popularity, quickly moving beyond its initial plan as an experiment. As of early 2016 it had over 3,000,000 downloads. The Redux repository on GitHub has over 50,000 stars, and Redux is now used by a number of companies including Exana, Patreon, and ClassPass.

Since the introduction of Redux, Context has been added to React. Context, like Redux, gives you a way to store and manage global state in your React applications. For projects with simpler global state requirements, Context has become a popular alternative to using Redux.

Context is built into React so there's no need to install an additional library as a dependency. Context is also simpler overall and generally requires less work to get up and running. All that being said, for projects with more sophisticated global state requirements, Redux remains a popular option. Redux offers greater flexibility with support for middleware and richer developer tools in the form of the Redux DevTools.

Vocabulary

Learning how to use Redux requires you to understand a fair amount of terminology. For now, don't worry about memorizing all of the following terms; it's good enough to just have a general awareness. You'll revisit each of these terms as you work your way through this lesson.

State

Ex: "*Redux is a state manager.*"

The *state* of a program means all the information stored by that program at a particular point in time. It is generally used to refer to the data stored by the program at a particular instance in time, as opposed to the logic of the program, which doesn't change over time. The job of Redux is to store the state of your app and make it available to entire app.

Store

Ex: "*Redux stores state in a single store.*"

The Redux store is a single JavaScript object with a few methods, including `getState`, `dispatch(action)`, and `subscribe(listener)`. Any state you want Redux to handle is held in the store.

Actions

Ex: "*The Redux store is updated by dispatching actions.*"

An action is a POJO (plain old JavaScript object) with a `type` property. Actions contain information that can be used to update the store. They can be *dispatched*, i.e. sent to the store, in response to user actions or AJAX requests. Typically Redux apps use functions called *action creators* that return actions. Action creators can take arguments which allow them to customize the data contained in the actions they generate.

Pure functions

Ex: "*Redux reducers are pure functions.*"

A function is pure if its behavior depends only its arguments and it has no side effects. This means the function can't depend on the value of any variables that

aren't passed to it as arguments, and it can't alter the state of the program or any variable existing outside itself. It simply takes in arguments and returns a value.

Reducer

Ex: "*Redux handles actions using reducers.*"

A reducer is a function that is called each time an action is dispatched. The reducer receives an action and the current state as arguments and returns an updated state.

Redux reducers are required to be pure functions of the dispatched action and the current state. This makes their behavior very predictable and allows their effects to potentially be reversed.

Middleware

Ex: "*You can customize your response to dispatched actions using middleware.*"

Middleware is an optional component of Redux that allows custom responses to dispatched actions. When an action is dispatched, it passes through each middleware that has been added to the state. The middleware can take some action in response and chose whether or not to pass the action on down the chain. Behind the scenes, the middleware actually replaces the dispatch method of the store with a customized version. There is a large ecosystem of existing middleware ready to be plugged into any Redux projects. One example is a logger that records each action before passing it on to the reducer. Perhaps the most common use for middleware is to dispatch asynchronous requests to a server.

Time traveling dev tools

Ex: "*Redux has time traveling dev tools.*"

Redux reducers are pure functions of the dispatched action and the current state. This means that if one were to store a list of the previous states over time and the actions that had been dispatched, one could retroactively cancel an action and recalculate the state as if that action had never been dispatched. This is precisely the functionality that the Redux DevTools provide. The dev tools can be added as middleware to any Redux project. They allow you to look back through the history of the state and toggle past actions on and off to see a live recalculation of the state. This ability to revert to a previous state is what is meant by time travel.

Thunks

Ex: *"Thunks are a convenient format for taking asynchronous actions in Redux."*

The traditional approach to middleware closely parallels the format of reducers. The actions being dispatched are POJOs with types and various middleware files are waiting to receive them. These files check the type of the action using a case statement just like reducers.

Thunks are an alternative approach. A thunk is a general concept in computer science referring to a function whose primary purpose is simply to call another function. In Redux a thunk action creator returns a function rather than an object. When they are dispatched, thunk actions are intercepted by a piece of middleware that simply checks if each action is a function. If it is, that function is called with the state and dispatch as arguments, otherwise it is passed on down the chain. Thunks are most commonly used to make asynchronous API requests.

What you learned

In this article, you learned what Redux is and where it came from. You also learned when it's appropriate to use Redux and some of the vocabulary terms used

by Redux.

See also...

The [official Redux documentation](#) is a great resource for learning more about Redux. To see who's using Redux, see this page on [StackShare](#).

Flux and Redux

Redux is an evolution of the concepts introduced by Flux. Having a general understanding of Flux will assist you in learning Redux.

When you finish this article, you should be able to:

- Describe the relationship between Redux and Flux
- Describe the three principles that Redux abides by
- Describe the Redux data cycle

What is Flux?

Flux is a front-end application architecture Facebook developed to use with React. Flux is not a library or framework. Flux is simply a pattern in which to structure one's application. It doesn't even need to be used with React! Flux provides unidirectional data flow, which affords more predictability than one might encounter when using other application design patterns.



Actions

An action begins the flow of data in Flux. An action is a simple object that at a minimum contains a `type`. An action's `type` indicates the type of change to be performed on the application's state. An action may contain additional data (the "payload") that's necessary for changing the application's former state to its next one.

Dispatcher

The dispatcher is a mechanism for distributing (or "dispatching") actions to a Flux application's store. The dispatcher is little more than a registry of callback functions into the store. Redux (the implementation of Flux we'll use at App Academy) consolidates the dispatcher into a single `dispatch()` function.

Store

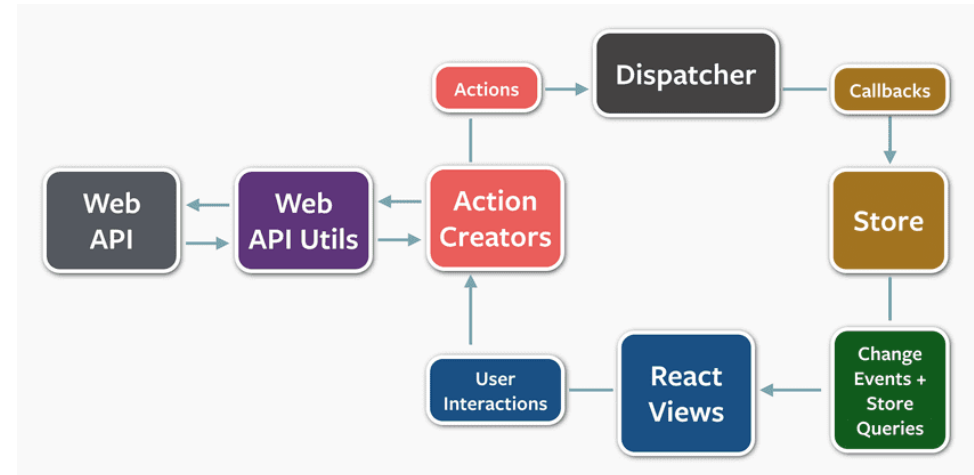
The store represents the entire state of the application. It's also responsible for updating the state of the application appropriately whenever it receives an action. It does so by registering with the dispatcher a callback function that receives an action. This callback function uses the action's type to invoke the proper function to change the application's state. After the store has changed state, it "emits a change," i.e. the store passes the new state to any views (explanation incoming) that have registered listeners (callbacks) to it.

View

A view is a unit of code that's responsible for rendering the user interface. To complete the Flux pattern, a view listens to change events emitted by the store. When a change to the application's data layer occurs, a view can respond appropriately, such as by updating its internal state and triggering a re-render.

A view can create actions itself, e.g. in user-triggered events. If a user marks a todo as complete, a view might call a function that would dispatch an action

to toggle the todo's state. Creating an action from the view turns our Flux pattern into a unidirectional loop.



Here the original action might (for example) result from an asynchronous request to fetch todos from the database with a success callback to dispatch our action to receive those todos and update the application's state accordingly. It's a common pattern in Flux to dispatch an action that populates the initial state of the application, with further modifications coming from the client.

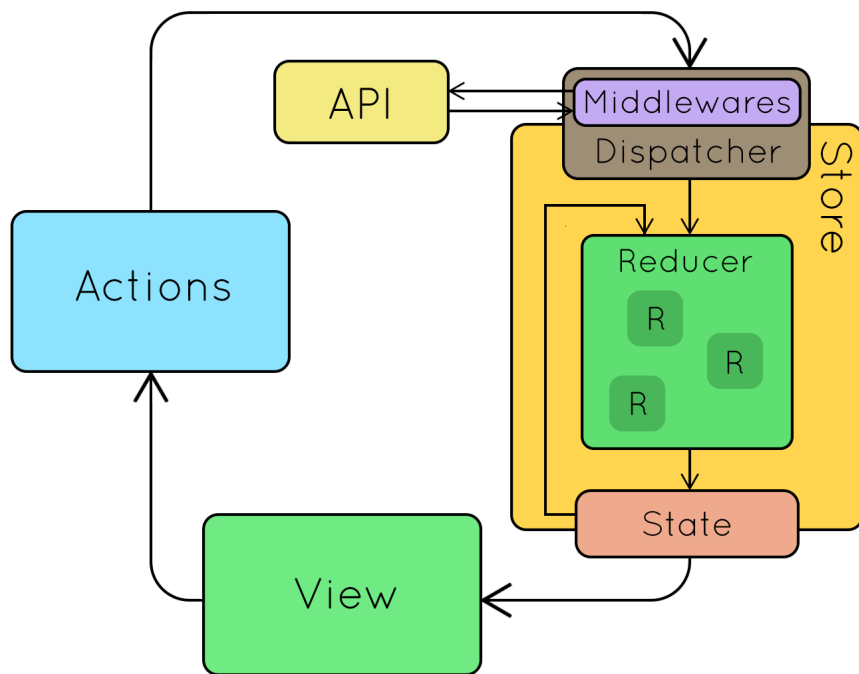
Redux

Redux is a library (distributed as an npm package) that facilitates a particular implementation of Flux. A Redux loop behaves slightly differently than a vanilla Flux loop, but the general concepts remain the same. Redux abides by three principles:

1. **Single Source of Truth:** The entire state of the application is stored in a single JavaScript object in a single store. This object is commonly

referred to as a “state tree” because its values often contain or are objects themselves.

2. **State is Read-Only:** The only way to change the state is to dispatch an action. This principle ensures that our Redux loop is never short-circuited and change of state remains single-threaded.
3. **Only Pure Functions Change State:** Pure functions known as “reducers” receive the previous state and an action and return the next state. They return new state objects instead of mutating previous state. Read [more](#) about what makes a function pure.



As you've probably already surmised, `React` will be our view layer.

Note: Middleware is an ecosystem of utilities that augments the functionality of `dispatch()`. Among other things, it allows for asynchronous

requests in a Redux application.

You'll learn more about each part in the Redux loop in this lesson.

What you learned

In this article, you learned about the relationship between Redux and Flux. You also learned about the three principles that Redux abides by and the Redux data cycle.

Store

The **store** is the central element of Redux's architecture. It holds the global **state** of an application. The store is responsible for updating the global state via its **reducer**, broadcasting state updates via **subscription**, and listening for **actions** that tell it when to update the state.

When you finish this article, you should be able to:

- Describe the role of the store in the Redux architecture
- Use the `createStore` method to create an instance of the Redux store
- Use the `store.dispatch` method to dispatch an action to trigger a state update
- Use the `store.subscribe` method to listen for state updates
- Use the `store.getState` method to get the current state

Creating the store

The `redux` library provides us with a `createStore()` method, which takes up to three arguments and returns a Redux store.

```
createStore(reducer, [preloadedState], [enhancer]);
```

- `reducer` (required) - A reducing function that receives the store's current state and incoming action, determines how to update the store's state, and returns the next state (more on this in a moment).
- `preloadedState` (optional) - An `object` representing any application state that existed before the store was created.
- `enhancer` (optional) - A `function` that adds extra functionality to the store.

You'll learn more about how to use the `preloadedState` and `enhancer` parameters later in this lesson. For now you'll focus on creating a store with just the single required `reducer` parameter.

Here is an example of how to create a store for a Fruit Stand application:

```
import { createStore } from 'redux';

const fruitReducer = (state = [], action) => {
  // TODO implement reducer
}

const store = createStore(fruitReducer);
```

A Redux application will typically only have a single store. You'll implement the `reducer` function in just a bit.

Store API

A Redux store is just an object that holds the application state, wrapped in a minimalist API. The store has three methods: `getState()`, `dispatch(action)`,

and `subscribe(callback)`.

Store methods

- `getState()` - Returns the store's current state.
- `dispatch(action)` - Passes an `action` into the store's `reducer` telling it what information to update.
- `subscribe(callback)` - Registers a callback to be triggered whenever the store updates. Returns a function, which when invoked, unsubscribes the callback function from the store.

Updating the store

Store updates can only be triggered by dispatching **actions**:

```
store.dispatch(action);
```

An `action` in Redux is just a plain object with:

- a `type` key indicating the action being performed, and
- optional payload keys containing any new information.

For example, the store for your Fruit Stand application would handle the inventory. You would use the following `addOrange` action to add an orange to the store's state. Notice how it has a `type` of 'ADD_FRUIT' and a `fruit` payload of 'orange':

```
const addOrange = {
  type: 'ADD_FRUIT',
  fruit: 'orange',
};
```

When `store.dispatch()` is called, the store passes its current `state`, along with the `action` being dispatched, to the `reducer`. The `reducer` function

takes the two arguments (`state` and `action`) and returns the next `state` . You'll read more about the `reducer` in just a bit, but for now, think of it as a Redux app's traffic cop, routing new information to its rightful place in the state.

A `reducer` for the Fruit Stand application looks like this:

```
const fruitReducer = (state = [], action) => {
  switch (action.type) {
    case 'ADD_FRUIT':
      return [...state, action.fruit];
    default:
      return state;
  }
};
```

The reducer's `state` parameter provides a default value; this is the **initial state** of our store prior to any actions. In this case, it's an empty array. In Redux, **the state is immutable**, so the reducer must return a **new array or object** whenever the state changes.

Now that you've defined your app's reducing function, you can now `dispatch` the `addOrange` action to the store:

```
console.log(store.getState()); // []
store.dispatch(addOrange);
console.log(store.getState()); // [ 'orange' ]
```

Subscribing to the store

Once the store has processed a `dispatch()` , it triggers all its subscribers. Subscribers are callbacks that can be added to the store via `subscribe()` .

You can define a callback `display` and subscribe it to the store:

```
const display = () => {
  console.log(store.getState());
};

const unsubscribeDisplay = store.subscribe(display);

store.dispatch(addOrange); // [ 'orange', 'orange' ]

// display will no longer be invoked after store.dispatch()
unsubscribeDisplay();

store.dispatch(addOrange); // no output
```

In the example above, the store processed the dispatched action and then called all of its subscribers in response. The only subscriber is your `display` callback which logs the current state when called.

Later in this lesson, you'll learn how to use the `store.subscribe()` method to connect a React component to the store so that it can listen for global state updates.

Reviewing a simple example

Later in this lesson, you'll see how to use Redux with React and how to organize your Redux code into separate modules, but for now to keep things as simple as possible, you'll put everything into a single file and use Node.js to run your application.

Here's an `app.js` file that brings together all of the above code snippets into a single example:

```
// app.js

const { createStore } = require('redux');

// Define the store's reducer.
const fruitReducer = (state = [], action) => {
  switch (action.type) {
    case 'ADD_FRUIT':
      return [...state, action.fruit];
    default:
      return state;
  }
};

// Create the store.
const store = createStore(fruitReducer);

// Define an 'ADD_FRUIT' action for adding an orange to the store.
const addOrange = {
  type: 'ADD_FRUIT',
  fruit: 'orange',
};

// Log to the console the store's state before and after
// dispatching the 'ADD_FRUIT' action.
console.log(store.getState()); // []
store.dispatch(addOrange);
console.log(store.getState()); // [ 'orange' ]

// Define and register a callback to listen for store updates
// and console log the store's state.
const display = () => {
  console.log(store.getState());
};
const unsubscribeDisplay = store.subscribe(display);

// Dispatch the 'ADD_FRUIT' action. This time the `display` callback
// will be called by the store when its state is updated.
store.dispatch(addOrange); // [ 'orange', 'orange' ]

// Unsubscribe the `display` callback to stop listening for store updates.
unsubscribeDisplay();
```

```
// Dispatch the 'ADD_FRUIT' action one more time
// to confirm that the `display` method won't be called
// when the store state is updated.
store.dispatch(addOrange); // no output
```

To run the above example, use npm to initialize the project (`npm init -y`) and to install Redux (`npm install redux`). Then use the command `node app.js` to run the example. You should see the following output:

```
[]
[ 'orange' ]
[ 'orange', 'orange' ]
```

What you learned

In this article, you learned about the role of the store in the Redux architecture. You saw how to use the `createStore` method to create a store instance, the `store.dispatch` method to dispatch an action to trigger a state update, the `store.subscribe` method to listen for state updates, and `store.getState` method to get the current state.

See also...

To learn more about the store, see the official [Redux documentation](#).

Reducers

As you saw in an earlier article, the Redux store has a **reducer** function for updating its state. The reducer function receives the current `state` and an `action`, updates the state appropriately based on the `action.type`, and returns the next state.

When you finish this article, you should be able to:

- Explain what a *reducer* is
- Use a `switch` statement within a reducer function to handle multiple action types
- Describe why it's important for a reducer to avoid mutating the current state when creating the next state

Updating the reducer to handle additional action types

Recall the reducer from the Fruit Stand application:

```
const fruitReducer = (state = [], action) => {
  switch (action.type) {
    case 'ADD_FRUIT':
      return [...state, action.fruit];
    default:
      return state;
  }
};
```

When the store initializes, it calls its reducer with an `undefined` `state`, allowing the reducer to dictate the store's initial state via the `state` parameter's default value.

The bulk of the reducer function then implements updates to the state. First, the reducer decides what logic to implement based on the `action.type` `switch`. Then, it creates and returns a new object representing the next state (after

processing the action) if any of the information needs to be changed. The `state` is returned unchanged if no cases match the `action.type`, meaning that the reducer doesn't *care* about that action (e.g. `{type: 'NEW_TRANSFORMERS_SEQUEL'}`).

In the above example, the reducer's initial state is set to an empty array (i.e. `[]`). The reducer returns a new array with `action.fruit` appended to the previous `state` if `action.type` is `'ADD_FRUIT'`. Otherwise, it returns the `state` unchanged.

Additional `case` clauses can be added to update the reducer to handle the following action types:

- `'ADD_FRUITS'` - Add an array of fruits to the inventory of fruits
- `'SELL_FRUIT'` - Remove the first instance of a fruit if available
- `'SELL_OUT'` - Someone bought the whole inventory of fruit! Return an empty array

```
const fruitReducer = (state = [], action) => {
  switch (action.type) {
    case 'ADD_FRUIT':
      return [...state, action.fruit];
    case 'ADD_FRUITS':
      return [...state, ...action.fruits];
    case 'SELL_FRUIT':
      const index = state.indexOf(action.fruit);
      if (index !== -1) {
        // remove first instance of action.fruit
        return [...state.slice(0, index), ...state.slice(index + 1)];
      }
      return state; // if action.fruit is not in state, return previous state
    case 'SELL_OUT':
      return [];
    default:
      return state;
  }
};
```

Reviewing how `Array#slice` works

If you don't regularly use the `Array#slice` method, the following expression might look odd at first glance:

```
[...state.slice(0, index), ...state.slice(index + 1)]
```

The `Array#slice` method returns a new array containing a shallow copy of the array elements indicated by the `start` and `end` arguments. The `start` argument is the index of the first element to include and the `end` argument is the index of the element to include up to (but not including). If the `end` argument isn't provided, all of the array elements up to the end of the array will be included. The original array will not be modified.

By combining two calls to the `Array#slice` method into a new array, a copy of an array can be created that omits an element at a specific index (`index`):

- `state.slice(0, index)` - Returns a new array containing the elements starting from index `0` up to `index`
- `state.slice(index + 1)` - Returns a new array containing the elements starting from `index + 1` (one past the index to omit the element at `index`) up through the last element in the array

Then the spread syntax is used to spread the elements in the slices into a new array.

Here's a complete example:

```
const fruits = ['apple', 'apple', 'orange', 'banana', 'watermelon'];

// The index of the 'orange' element is 2.
const index = fruits.indexOf('orange');

// `...fruits.slice(0, index)` returns the array ['apple', 'apple']
// `...fruits.slice(index + 1)` returns the array ['banana', 'watermelon']
// The spread syntax combines the two array slices into the array
// ['apple', 'apple', 'banana', 'watermelon']
const newFruits = [...fruits.slice(0, index), ...fruits.slice(index + 1)];
```

This approach to removing an element from an array is just one way to complete the operation without modifying or mutating the original array.

Avoiding state mutations

Inside a Redux reducer, you must never mutate its arguments (i.e. `state` and `action`). **Your reducer must return a new object if the state changes.** [Here's why.](#)

Here's an example of a *bad* reducer which mutates the previous state.

```
const badReducer = (state = { count: 0 }, action) => {
  switch (action.type) {
    case 'INCREMENT_COUNTER':
      state.count++;
      return state;
    default:
      return state;
  }
};
```

And here's an example of a good reducer which uses `Object.assign` to create a shallow duplicate of the previous `state` :

```
const goodReducer = (state = { count: 0 }, action) => {
  switch (action.type) {
    case 'INCREMENT_COUNTER':
      const nextState = Object.assign({}, state);
      nextState.count++;
      return nextState;
    default:
      return state;
  }
};
```

What you learned

In this article, you learned about reducers and how to use a `switch` statement within a reducer function to handle multiple action types. You also learned why it's important for a reducer to avoid mutating the current state when creating the next state.

See also...

To learn more about reducers, see the official [Redux documentation](#).

Actions

As you've already learned from an earlier article, **actions** are the only way to trigger changes to the store's state. Remember, actions are simple POJOs with a mandatory `type` key and optional payload keys containing new information. They get sent using `store.dispatch()` and are the primary drivers of the Redux loop.

When you finish this article, you should be able to:

- Write an action creator function to facilitate in the creation of action objects
- Use constants to define action types to prevent simple typos in action type string literals

Using action creators

When an action is dispatched, any new state data must be passed along as the **payload**. The example below passes a payload key of `fruit` with the new state data, 'orange':

```
const addOrange = {
  type: 'ADD_FRUIT',
  fruit: 'orange',
};

store.dispatch(addOrange);
console.log(store.getState()); // [ 'orange' ]
```

However, when these action payloads are generated dynamically, it becomes necessary to extrapolate the creation of the action object into a function. These functions are called **action creators**. The JavaScript objects they return are the **actions**. To initiate a dispatch, you pass the result of calling an action creator to `store.dispatch()`.

For example, an action creator function to create 'ADD_FRUIT' actions looks like this:

```
const addFruit = (fruit) => {
  return {
    type: 'ADD_FRUIT',
    fruit,
  };
};
```

You can also rewrite the above arrow function to use an implicit return value:

```
const addFruit = (fruit) => ({
  type: 'ADD_FRUIT',
  fruit,
});
```

While either approach for defining action creators using arrow functions works, the latter approach of using an implicit return value makes it more difficult to debug the Redux cycle (you'll see why later in this lesson).

Now we can add any `fruit` to the store using our action creator `addFruit(fruit)`, instead of having to define an action object for each fruit:

```
store.dispatch(addFruit('apple'));
store.dispatch(addFruit('strawberry'));
store.dispatch(addFruit('lychee'));
console.log(store.getState()); // [ 'orange', 'apple', 'strawberry', 'lychee' ]
```

Preventing typos in action type string literals

Update your actions to include `'ADD_FRUIT'`, `'ADD_FRUITS'`, `'SELL_FRUIT'`, and `'SELL_OUT'`:

```
const ADD_FRUIT = 'ADD_FRUIT';
const ADD_FRUITS = 'ADD_FRUITS';
const SELL_FRUIT = 'SELL_FRUIT';
const SELL_OUT = 'SELL_OUT';

const addFruit = (fruit) => ({
  type: ADD_FRUIT,
  fruit,
});

const addFruits = (fruits) => ({
  type: ADD_FRUITS,
  fruits,
});

const sellFruit = (fruit) => ({
  type: SELL_FRUIT,
  fruit,
});

const sellOut = () => ({
  type: SELL_OUT,
});
```

Notice that constants were used for all of the fruit action types. This prevents simple typos in the reducer's case clauses (i.e. `'ADD_FRIUT'`) from unexpectedly not matching the appropriate action type (i.e. `'ADD_FRUIT'`). Creating constants for the action type string literals ensures that an error is thrown if the constant name is mistyped.

Reviewing a completed Fruit Stand example

To review and run a completed Fruit Stand example application, clone the [redux-fruit-stand-examples](#) repo.

After cloning the repo, open a terminal and browse to the `fruit-stand-redux` folder. Run the command `npm install` to install the project's dependencies.

Then use the command `node app.js` to run the Fruit Stand application. You should see the following output:

```
Default Redux Store (empty fruit list):
[]
Redux Store:
[ 'orange', 'apple' ]
Redux Store:
[ 'orange', 'apple', 'orange', 'lychee', 'grapefruit' ]
Updated Redux Store:
[ 'orange', 'orange', 'lychee', 'grapefruit' ]
Reset Redux Store (empty fruit list):
[]
```

The `reduxSAR.js` file contains the action types, reducer, store, and action creator functions. The `app.js` file contains the code that interacts with the Redux store. The `appWithSubscription.js` file also contains code that interacts with the store but subscribes a callback function (using the `store.subscribe` method) to listen for and log state updates to the console.

What you learned

In this article, you learned how to write an action creator function to facilitate in the creation of action objects. You also learned how to use constants to define action types to prevent simple typos in action type string literals.

See also...

To learn more about actions, see the official [Redux documentation](#).

Debugging Arrow Functions

Arrow functions are ubiquitous in React and Redux. Understanding how to use `debugger` statements with arrow functions is necessary to be able to effectively debug the Redux cycle.

When you finish this article, you should be able to:

- Describe why `debugger` statements can't be used with arrow functions that have an implicit return value
- Rewrite an arrow function with an implicit return value to use an explicit return statement so that a `debugger` statement can be added

Understanding the limitations of implicit return values

Here's an example of a Redux action creator that's defined using an arrow function with an implicit return value:

```
const addFruit = (fruit) => ({
  type: 'ADD_FRUIT',
  fruit,
});
```

While using an arrow function with an implicit return value allows you to concisely define `addFruit`, it's difficult to debug. Suppose you want to use a `debugger` statement to stop execution within `addFruit` to inspect the value of the `fruit` parameter. You **can't** do this:


```
const addFruit = (fruit) => ({
  debugger
  type: 'ADD_FRUIT',
  fruit,
});
```

`{ type: 'ADD_FRUIT', fruit }` is an object, and you can't put a `debugger` statement inside of an object. But you also **can't** do this:

```
const addFruit = (fruit) => (
  debugger
  {
    type: 'ADD_FRUIT',
    fruit,
  }
);
```

The parentheses after the fat arrow (`=>`) are used to indicate that the object `{ type: 'ADD_FRUIT', fruit }` should be implicitly returned. As a result, the above won't work, because we can't put a debugger inside of a return statement.

Rewriting an arrow function to use an explicit return statement

To put a `debugger` statement inside of the `addFruit` action creator function, you first need to convert it into an arrow function with an explicit return statement:

```
const addFruit = (fruit) => {
  return {
    type: 'ADD_FRUIT',
    fruit,
  };
};
```

Now, finally, you can put the `debugger` statement before the `return` statement:

```
const addFruit = (fruit) => {
  debugger;
  return {
    type: 'ADD_FRUIT',
    fruit,
  };
};
```

If you want to avoid having to do this over and over again as you're debugging your arrow functions, you can make it a habit to write all of your arrow functions with explicit return statements. Do be aware, however, that writing arrow functions with implicit return values is a common convention in Redux and you will see it often out in the wild.

What you learned

In this article, you learned why `debugger` statements can't be used with arrow functions that have an implicit return value. You also learned how to rewrite an arrow function with an implicit return value to use an explicit return statement so that a `debugger` statement can be added.

Redux To-do List Project

At this point, you understand how to perform CRUD operations with a backend API. You also know how to perform CRUD operations by creating a user interface with React, managing state with React Context, and storing persistent data in local storage. It's time to create a simple Node to-do list application that utilizes the Redux library and runs in your terminal! This project is intended as a way

to practice the basics of Redux before learning how to use Redux within a React application.

In today's project, you will:

- Generate a Redux **store** by using the `createStore` method from the Redux library
- Set up a **reducer** to direct different action types to interact with the Redux store in different ways
- Set up **actions** to create a task, delete a task, and reset the task list
- Use the `store.getState` method to access the data stored in the Redux store
- Use the `store.dispatch` method to **dispatch** actions to the Redux store
- Use the `store.subscribe` method to subscribe to Redux store changes
- Use the VS Code debugger to follow the Redux cycle

Phase 1: Create Redux store, actions, and reducer

Run the following commands in your terminal to create a new project directory, generate a `package.json` file, and install `redux` as a dependency:

```
mkdir redux-todos && cd redux-todos
npm init -y && npm install redux
```

Now it's time to create your first Redux project! Create a `reduxStoreActionReducer.js` file and import the `createStore` method from Redux. You'll need to use CommonJS module syntax (`module.exports` and `require`) to be able to run the project within the Node environment:

```
const { createStore } = require('redux');
```

Store

You'll use the `createStore` method to generate your Redux store by invoking it with a reducer. As a reminder, each application should only have one Redux store where all of an application's state is managed. This is unlike using React Context, where a single React application can utilize multiple contexts.

Conceptually speaking, you can think of the reducer as a function that helps manage the Redux store by routing actions based on their `type` attribute. Based on the official Redux documentation on the `createStore` method and its parameters, a reducer is a "reducing function that returns the next state tree, given the current state tree and an action to handle."

Now that we've gone over a conceptual overview of reducers, let's create a `tasksReducer` to manage the to-do list tasks in your Redux store!

Define the `tasksReducer` function and have it take in the Redux store's `state` and an `action` as parameters. You'll want the `state` to default to an empty (`[]`) if the `tasksReducer` is invoked without a state. Since you invoke the `createStore` method with the `tasksReducer` , you'll need to define the `tasksReducer` before invoking the `createStore` method to generate the Redux store.

Note: this project uses an array to store the tasks instead of an object to make it easier to delete a positional task, since there is no user interface that will expose the task ID needed to dispatch task deletion. In the case of this project, a `taskId` will refer to the index of a task in the array.

```
const tasksReducer = (state = [], action) => {
  // TODO: Set up switch statement to manage actions based on type
};

const store = createStore(tasksReducer);
```

The underlying code in the store returned by the `createStore` method will automatically invoke the `tasksReducer` function whenever an action is

dispatched. Speaking of actions, let's set up the `createTask` , `deleteTask` , and `resetTaskList` actions before finishing the `tasksReducer` function that conceptually *routes* action objects (think of how it makes more sense to create a component before setting up a `<Route>` for it).

You'll finish defining the `tasksReducer` function after creating the `createTask` , `deleteTask` , and `resetTaskList` actions that the reducer manages. You'll set up your actions below the line of code that sets up the Redux `store` .

Actions

Let's set up the `createTask` action creator! You'll want your `createTask` action creator to return an action with the following shape:

```
{
  type: 'CREATE_TASK',
  taskMessage: 'walk dog',
}
```

As a reminder, an **action creator** is simply a function that returns an **action** which is a POJO (plain old JavaScript object) that defines a `type` key and optional payload keys. Have your `createTask` function take in a `taskMessage` as a parameter:

```
const createTask = (taskMessage) => {
  // TODO: Return POJO with `type` property and function's argument (`taskMessage`);
};
```

Now you'll want to set up the function's return statement to return the action POJO. The POJO should have a `type` property. The `type` property will be how the `tasksReducer` will decipher different types of actions to update the Redux

store's state in different ways. The action will also have a payload key set to the function's argument (`taskMessage`).

In this case, the `createTask` function has a `taskMessage` parameter, so the action POJO will have a `type` property set to the string `CREATE_TASK` , as well as a `taskMessage` property set to the `taskMessage` parameter value.

```
const createTask = (taskMessage) => {
  return {
    type: 'CREATE_TASK',
    taskMessage: taskMessage,
  };
};
```

It is best practice to use constants for action types, instead of string literals. Since the reducer depends on the action's `type` to decipher different types of actions, a typo in the reducer or action specifying the type will go unseen. For example, imagine if the reducer needs an action with the type `CREATE_TASK` to perform the create operation, but there is a typo making the reducer listen for the type `'CREATE_TSAK'` instead. When an action of type `CREATE_TASK` is dispatched, it will never be evaluated by the reducer listening for an action of type `'CREATE_TSAK'` . Creating constants for string literals ensures that an error will be thrown for action type typos.

Define a constant for the `CREATE_TASK` string. Make sure to define the constant before defining your `tasksReducer` , otherwise you'll receive `ReferenceError: CREATE_TASK is not defined` when you dispatch an action. At this point, your file should look something like this:

```
const { createStore } = require('redux');

const CREATE_TASK = 'CREATE_TASK';

const tasksReducer = (state = [], action) => {
  // TODO: Set up switch statement to manage actions based on type
};

const store = createStore(tasksReducer);

const createTask = (taskMessage) => {
  return {
    type: CREATE_TASK,
    taskMessage: taskMessage,
  };
};
```

You can also have the function implicitly return by removing the function's curly braces and wrapping the action object's curly braces with parentheses. The `createTask` code below has the exact same functionality as the code above:

```
const createTask = (taskMessage) => ({
  type: CREATE_TASK,
  taskMessage,
});
```

Although the code looks shorter and cleaner, note that you are unable to use the [debugger statement](#) to debug a function when it is implicitly returning. You can use the VS Code debugger to set a breakpoint, but you won't be able to set a breakpoint with the `debugger` statement.

When working on your React-Redux projects in the future, it'll be helpful to keep the `return` statement so you can easily place a breakpoint with the `debugger` statement to debug the action creator function and make sure a specific action is actually being dispatched.

Now that you have set up a `createTask` action creator, follow the same pattern to set up a `deleteTask` action creator that takes in a `taskId`. The `resetTaskList` action creator will be a little different. You'll still follow the pattern of setting a `type` for the action, but the `resetTaskList` function will not take any parameters. The action will simply have a `type` property and a `emptyTaskList` property set to an empty array:

```
const resetTaskList = () => {
  return {
    type: RESET_TASK_LIST,
    emptyTaskList: [],
  };
};
```

Reducer

It's time to circle back and finish implementing the `tasksReducer`! Begin by setting up a `switch` statement that evaluates a case statement based on the `action.type`.

```
const tasksReducer = (state = [], action) => {
  switch (action.type) {
    // TODO: Set up switch case for `createTask` action
    // TODO: Set up switch case for `deleteTask` action
    // TODO: Set up switch case for `resetTaskList` action
    // TODO: Set up default switch case
  }
};
```

Let's begin by setting up the default switch case. By default, you always want to return the default `state` argument passed into the reducer:

```
const tasksReducer = (state = [], action) => {
  switch (action.type) {
    // TODO: Set up switch case for `createTask` action
    // TODO: Set up switch case for `deleteTask` action
    // TODO: Set up switch case for `resetTaskList` action
    default:
      return state;
  }
};
```

Now let's set up the `case` statements for each action type. You have three task actions, so you'll set up three case statements. As a reminder, you set up the action types with constants to make sure typos in the reducer's `case` statements throw an error. Use the constants in the switch statement:

```
const tasksReducer = (state = [], action) => {
  switch (action.type) {
    case CREATE_TASK:
      // TODO: Define what happens when a `createTask` action is dispatched
    case DELETE_TASK:
      // TODO: Define what happens when a `deleteTask` action is dispatched
    case RESET_TASK_LIST:
      // TODO: Define what happens when a `resetTaskList` action is dispatched
    default:

      return state;
  }
};
```

As a reminder, the reducer function is called every time an action is dispatched. This means that the `tasksReducer` function will be invoked whenever an action is dispatched. It's now time to write code to handle each specific action type and define what happens when actions of different types are dispatched!

CREATE_TASK case statement

Under the case statement for `CREATE_TASK`, you'll want to return an updated version of the reducer's state tree, with the new task. As a reminder, the Redux store's state should be immutable - this means you should never mutate the `state` array directly.

Begin by generating a `newTask` object based on the action's `taskMessage` property:

```
const newTask = {
  message: action.taskMessage,
};
```

Since you don't want to directly mutate the `state` array, you don't want to push the `newTask` directly into the `state` array. Instead, you can use spread syntax to return an updated state with `newTask` set as the last array element in a new array:

```
case CREATE_TASK:
  const newTask = {
    message: action.taskMessage,
  };
  return [...state, newTask];
```

DELETE_TASK case statement

Now let's define what happens when a `DELETE_TASK` action is dispatched. As a reminder, the `deleteTask` action creator function takes in a `taskId` that actually references a task element's index in the `state` array. In the `DELETE_TASK` case statement, you'll use the index (the action's `taskId` property) and the native [Array.slice](#) method to return a copy of the state that excludes the task to delete:

```
case DELETE_TASK:
  const idx = action.taskId;
  return [...state.slice(0, idx), ...state.slice(idx + 1)];
```

Using spread syntax and non-mutative methods are one of the many immutable update patterns you can use to update state without directly mutating it. Feel free to visit the official Redux documentation to view more [immutable update patterns](#).

RESET_TASK_LIST case statement

Now let's define what happens when a `RESET_TASK_LIST` action is dispatched. As a reminder, the action has an `emptyTaskList` property that is an empty array, similar to the default state set by the reducer. You can simply return the `emptyTaskList` array to update the Redux store's state. The `emptyTaskList` will replace the `state` array entirely.

```
case RESET_TASK_LIST:
  return action.emptyTaskList;
```

Phase 2: Testing

Start by creating an `app.js` file and importing `store`, `createTask`, `deleteTask`, and `resetTaskList` from the `reduxStoreActionReducer.js` file. Since this project is running within the native Node environment, you'll need to use CommonJS module syntax (`require` and `module.exports`) to manage imports and exports within the project:

```
const {
  store,
  createTask,
  deleteTask,
  resetTaskList,
} = require('./reduxStoreActionReducer');
```

Before you begin dispatching actions to test the actions and reducer you have created, you'll need to set up a way to log the Redux store and view its current state. You can use the `store.getState` method to access the Redux store's current state and simply console log the Redux store's state that was retrieved. To make your logging more clear, you can even add a message labeling the status of the store logged. After the imports at the top of your `app.js` file, log the initial state of the Redux store (an empty task list) with the following `console.log` statements:

```
console.log('Default Redux Store (empty task list):');
console.log(store.getState());
```

At this point, take a moment to run your Node application by running the follow terminal statement from the root of the project directory:

```
node app.js
```

You should see the following output in your terminal:

```
Default Redux Store (empty task list):
[]
```

Notice how you are currently using `console.log` statements to test and debug your code. Later in the project, you'll get a chance to investigate your code with the VS Code debugger to gain more context and insight about your code and its variable values.

Dispatch the `CREATE_TASK` action

Now you can test whether you can actually create a task by using the `store.dispatch` method to dispatch the `CREATE_TASK` action. Invoke the `createTask` action creator function with a task message, and then dispatch the invoked action. As a reminder, dispatching the action will "send" it through the reducer (in this case, the `tasksReducer`) and determine what operation to perform based on the action's `type` property.

```
store.dispatch(createTask('walk dog'));
store.dispatch(createTask('feed cat'));
store.dispatch(createTask('talk to bird'));
store.dispatch(createTask('watch goldfish'));

console.log('Redux Store:');
console.log(store.getState());
```

Now run your application with `node app.js` and you should see the following output in your terminal:

```
Default Redux Store (empty task list):
[]
Redux Store:
[ { message: 'walk dog' },
  { message: 'feed cat' },
  { message: 'talk to bird' },
  { message: 'watch goldfish' } ]
```

That may have seemed like magic for now, but at the end of the project, you'll walk-through your project's code step-by-step to view what is really happening with the Redux cycle. For now, focus on understanding the idea of *dispatching an action*.

Dispatch the `DELETE_TASK` action

Now you can add the following code after your `CREATE_TASK` dispatch calls to dispatch a `DELETE_TASK` action and log the Redux store's updated state:

```
store.dispatch(deleteTask(0));
store.dispatch(deleteTask(1));

console.log('Updated Redux Store:');
console.log(store.getState());
```

Run your application with `node app.js` and you should see the following output in your terminal:

```
Default Redux Store (empty task list):
[]
Redux Store:
[ { message: 'walk dog' },
  { message: 'feed cat' },
  { message: 'talk to bird' },
  { message: 'watch goldfish' } ]
Updated Redux Store:
[ { message: 'feed cat' }, { message: 'watch goldfish' } ]
```

Dispatch the `RESET_TASK_LIST` action

Lastly, take a moment to test the dispatching of the `RESET_TASK_LIST` action and log the updated state by adding the following code after the `DELETE_TASK` dispatch calls:

```
store.dispatch(resetTaskList());
console.log('Reset Redux Store (empty task list):');
console.log(store.getState());
```

If you run your application with `node app.js` and you should see the following output in your terminal:

```
Default Redux Store (empty task list):
[]
Redux Store:
[ { message: 'walk dog' },
  { message: 'feed cat' },
  { message: 'talk to bird' },
  { message: 'watch goldfish' } ]
Updated Redux Store:
[ { message: 'feed cat' }, { message: 'watch goldfish' } ]
Reset Redux Store (empty task list):
[]
```

Subscribe to Redux store updates

Instead of having multiple console log statements to log `store.getState()`, you can have your store subscribe to changes in the state with the `store.subscribe` method. Create a new `appWithSubscription.js` file with the following code to view how you can invoke `store.subscribe` so that the state is logged anytime the store is updated (i.e. anytime an action is dispatched).

```
// ./appWithSubscription.js

const {
  store,
  createTask,
  deleteTask,
  resetTaskList,
} = require('./reduxStoreActionReducer');

console.log('Default Redux Store (empty task list):');
console.log(store.getState());

store.subscribe(() => console.log(store.getState()));

console.log('Task creation actions');
store.dispatch(createTask('walk dog'));
store.dispatch(createTask('feed cat'));
store.dispatch(createTask('talk to bird'));
store.dispatch(createTask('watch goldfish'));

console.log('Task deletion actions');
store.dispatch(deleteTask(0));
store.dispatch(deleteTask(1));

console.log('Task reset action');
store.dispatch(resetTaskList());
```

Run your application with `node appWithSubscription.js` and you should see the following output. Notice how the state was logged four times under "Task creation actions" because of how four actions were dispatched, and the state was logged twice after "Task deletion actions" because of how two actions were dispatched.


```

Default Redux Store (empty task list):
[]
Task creation actions:
[ { message: 'walk dog' } ]
[ { message: 'walk dog' }, { message: 'feed cat' } ]
[ { message: 'walk dog' },
  { message: 'feed cat' },
  { message: 'talk to bird' } ]
[ { message: 'walk dog' },
  { message: 'feed cat' },
  { message: 'talk to bird' },
  { message: 'watch goldfish' } ]
Task deletion actions:
[ { message: 'feed cat' },
  { message: 'talk to bird' },
  { message: 'watch goldfish' } ]
[ { message: 'feed cat' }, { message: 'watch goldfish' } ]
Task reset action:
[]

```

Debug to follow the Redux cycle

Now that you've used `console.log` statements to thoroughly investigate your project, you can set up the VS Code debugger and add some breakpoints to follow the Redux cycle. You'll use the breakpoints to investigate how invoking `store.dispatch` with the result from an action creator function directs the action to a specific switch case in the `tasksReducer` function. Start by creating a `.vscode` directory and a `launch.json` file to configure the VS Code debugger:

```

mkdir .vscode && cd .vscode
touch launch.json

```

Paste the following configuration into your `launch.json` file:

```

{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "skipFiles": [
        "<node_internals>/**"
      ],
      "program": "${workspaceFolder}/app.js"
    }
  ]
}

```

As a reminder, VS Code can also auto-generate the `.vscode` directory and the `launch.json` for you. Feel free to visit the VS Code documentation for more on [debugging with VS Code](#).

You can follow your code when an action object is dispatched by setting breakpoints. You can also examine the action's attributes by setting a breakpoint in the reducer. In the reducer, make sure to set the breakpoint within a `case` statement. For example, if you set a breakpoint in the `switch` statement, but *outside* of a `case` statement, you won't ever hit the breakpoint set.

```

// Example of bad `debugger` placement that will not work!

switch (action.type) {
  debugger;
  case CREATE_TASK:
    const newTask = {
      message: action.taskMessage,
    };
    return [...state, newTask];
}

```

Take a moment to use the `debugger` keyword to set breakpoints in the `createTask` action creator and `CREATE_TASK` case statement in your `reduxStoreActionReducer.js` file:

```
const tasksReducer = (state = [], action) => {
  switch (action.type) {
    case CREATE_TASK:
      debugger
      const newTask = {
        message: action.taskMessage,
      };
      return [...state, newTask];
    case DELETE_TASK:
      const idx = action.taskId;
      return [...state.slice(0, idx), ...state.slice(idx + 1)];
    case RESET_TASK_LIST:
      return action.emptyTaskList;
    default:
      return state;
  }
};
```

```
const createTask = (taskMessage) => {
  debugger
  return {
    type: CREATE_TASK,
    taskMessage,
  };
};
```

Now you'll be able to pause the running of your code to examine variables in the environment and view step-by-step updates to the Redux store's `state` as your code is evaluated.

Open `app.js` as the active file in your VS Code workspace. Press `F5` and select `Node.js` as your environment - VS Code will try to run your currently active file in debug mode. This will allow you to follow each dispatched action and watch how each dispatched action updates the Redux store's state.

As you step through each dispatched action, you'll notice that there really is a *cycle*: an action is generated, then the action is dispatched to go through a reducer, and then the store is updated.

Here is a quick breakdown of what happens with the `CREATE_TASK` action is dispatched, to guide the navigation of using VS Code debugger to investigate the Redux cycle:

1. The `createTask` action creator function is invoked with the string `'walk dog'`.
2. The `createTask` function returns a POJO (known as an "action") with a `type` attribute and `taskMessage` properties. The POJO is structured like this:

```
{
  type: 'CREATE_TASK',
  taskMessage: 'walk dog',
}
```

3. The `store.dispatch` method is invoked to dispatch the action POJO and invoke the `tasksReducer` function. Since the POJO has a type of `CREATE_TASK`, the case statement for `CREATE_TASK` will be evaluated:

```
case CREATE_TASK:
  const newTask = {
    message: 'walk dog', // This is `action.taskMessage`
  };
  return [...state, newTask];
```

4. The store's state is updated to be the new state returned by the reducer (`[...state, newTask]`).

Congratulations! You have just created your first Redux store, reducer, and actions. You also learned more about what a reducer is doing by investigating with console log statements and debugging the project with the VS Code debugger.

WEEK-15 DAY-2

React + Redux

Redux Learning Objectives: Part 2

To keep things as simple as possible when initially learning Redux, you started with using Redux independent of React. Now it's time to learn how to use Redux within a React application!

After reading and practicing how to use Redux with React, you should be able to:

- Add Redux actions, reducer(s), and a store to a React project
 - Update a React class component to subscribe to a Redux store to listen for state changes
 - Update a React component to dispatch actions to a Redux store
 - Define multiple reducers to manage individual slices of state
 - Use the Redux `combineReducers` method to combine multiple reducers into a single root reducer
 - Update a reducer to delegate a state update to a subordinate reducer
 - Describe how container components differ from presentational components
 - Write a container component to handle all of the Redux store interaction for one or more presentational components
 - Use `Object.freeze` to prevent the current state within a reducer from being mutated
 - Create a Redux store with preloaded state
-

Using Redux with React

To keep things as simple as possible when initially learning Redux, you started with using Redux independent of React. Now it's time to learn how to use Redux within a React application!

When you finish this article, you should be able to:

- Add Redux actions, reducer(s), and a store to a React project
- Update a React class component to subscribe to a Redux store to listen for state changes
- Update a React component to dispatch actions to a Redux store

Integrating Redux into a React application

The techniques shown in this article for integrating Redux into a React application, is just one step in your journey to learn Redux. As you work your way through this lesson, you'll learn how to improve upon these techniques to improve the organization of your code, the design of your components, and the overall performance of your application.

In general, the steps to integrate Redux into an existing React application are:

- Set up Redux
 - Install the `redux` npm package
 - Define your actions
 - Define your reducer(s)
 - Create your store
- Update components
 - Use `store.subscribe` to listen for state updates
 - Call `store.getState` to retrieve state for rendering
 - Call `store.dispatch` to dispatch actions to the store

Note: You'll start with writing all of the code to interact with the store within each component that needs to render state from the store or to dispatch actions. Later on, you'll learn how to improve the overall design of your application by using container components. Eventually, you'll learn how to use the [React-Redux](#) library's `connect` method to avoid writing container components by hand.

Organizing your Redux code

Instead of placing all of your Redux related code into a single file, you'll separate your store, reducer, and actions into their own files.

There are a variety of acceptable ways to organize your Redux code within a React project. When starting out with using Redux, organizing your code by type (i.e. separate files or folders for the store, reducers, and actions) often feels natural and makes it easy to find the file that you need to make a change to. As your projects increase in size and complexity, you might find that organizing your files by feature (i.e. locating all the files related to a feature inside of a single folder) will keep you from searching and jumping around a project that contains hundreds of files.

Note: How a project is organized is highly dependent upon who is working on the project. It's also not unusual for the organization of a project to evolve and change throughout its lifetime. Don't struggle too much with deciding on an approach when getting starting a new project. Pick an approach and move on to getting work done!

Following along

If you'd like to follow along, clone the [react-fruit-stand-with-react-starter](#) repo.

After cloning the repo, open a terminal and browse to the `starter` folder within the repo. Run the command `npm install` to install the project's dependencies (the `redux` package is already listed as a dependency). Then use the command `npm start` to run the Fruit Stand application.

This Fruit Stand example application is a React application created by the Create React App tooling. When running the application using `npm start`, the application should automatically open in your default browser. If it doesn't, you can manually browse to `http://localhost:3000/` to view the application.

Adding the actions

Within the React project's `src` folder, add a folder named `actions`. Within that folder, add a file named `fruitActions.js` containing the following code:

```
// ./src/actions/fruitActions.js

export const ADD_FRUIT = 'ADD_FRUIT';
export const ADD_FRUITS = 'ADD_FRUITS';
export const SELL_FRUIT = 'SELL_FRUIT';
export const SELL_OUT = 'SELL_OUT';

export const addFruit = (fruit) => ({
  type: ADD_FRUIT,
  fruit,
});

export const addFruits = (fruits) => ({
  type: ADD_FRUITS,
  fruits,
});

export const sellFruit = (fruit) => ({
  type: SELL_FRUIT,
  fruit,
});

export const sellOut = () => ({
  type: SELL_OUT,
});
```

Adding the reducer

Within the React project's `src` folder, add a folder named `reducers`. Within that folder, add a file named `fruitReducer.js` containing the following code:

```
// ./src/components/fruitReducer.js

import {
  ADD_FRUIT,
  ADD_FRUITS,
  SELL_FRUIT,
  SELL_OUT,
} from '../actions/fruitActions';

const fruitReducer = (state = [], action) => {
  switch (action.type) {
    case ADD_FRUIT:
      return [...state, action.fruit];
    case ADD_FRUITS:
      return [...state, ...action.fruits];
    case SELL_FRUIT:
      const index = state.indexOf(action.fruit);
      if (index !== -1) {
        // remove first instance of action.fruit
        return [...state.slice(0, index), ...state.slice(index + 1)];
      }
      return state; // if action.fruit is not in state, return previous state
    case SELL_OUT:
      return [];
    default:
      return state;
  }
};

export default fruitReducer;
```

Adding the store

Within the React project's `src` folder, add a file named `store.js` containing the following code:

```
// ./src/store.js

import { createStore } from 'redux';
import fruitReducer from './reducers/fruitReducer';

const store = createStore(fruitReducer);

export default store;
```

Writing Redux aware React components

Remember that the integration techniques shown in this article are just a starting point with using Redux with React components. As you work your way through this lesson, you'll learn how to improve upon these techniques.

Listening for state changes

Components that need to render state from the store can use the `store.subscribe` method to subscribe to listen for state updates. When a state update occurs, the `forceUpdate` method is called to render the component. Within the component's `render` method, the `store.getState` method is called to retrieve the current state. This approach ensures that whenever state is updated in the store (after the reducer has processed a dispatched action), the component will retrieve and render the updated state.

Note: Calling `forceUpdate` causes `render` to be called without first calling `shouldComponentUpdate`. Child components will go through their normal lifecycle, including calling `shouldComponentUpdate` to determine if the child component should render. While this pattern works, it's a rather blunt instrument for complex components, since re-rendering a parent causes re-rendering of all its children. Later in this lesson, you'll learn how the [React-Redux](#) library solves this problem.

The `componentDidMount` and `componentWillUnmount` class component lifecycle methods can be used to ensure that the component *subscribes* to the store when it's mounted and *unsubscribes* from the store when the component is about to be unmounted:

```
// ./src/components/FruitList.js

import React from 'react';
import store from '../store';

class FruitList extends React.Component {
  componentDidMount() {
    this.unsubscribe = store.subscribe(() => {
      this.forceUpdate();
    });
  }

  componentWillUnmount() {
    if (this.unsubscribe) {
      this.unsubscribe();
    }
  }

  render() {
    const fruit = store.getState();

    return (
      <div>
        {fruit.length > 0
          ? <ul>{fruit.map((fruitName, index) => <li key={index}>{fruitName}</li>
            : <span>No fruit currently in stock!</span>
          }
        </div>
      );
    }
  }

  export default FruitList;
```

Dispatching actions

Updating a component to dispatch an action to the store is a bit simpler overall than listening for and rendering state updates. You just need to import the appropriate action creator function and use the `store.dispatch` method within a event handler to dispatch the action:

```
// ./src/components/FruitQuickAdd.js

import React from 'react';
import store from '../store';
import { addFruit } from '../actions/fruitActions';

class FruitQuickAdd extends React.Component {
  addFruitClick = (event) => {
    const fruit = event.target.innerText;
    store.dispatch(addFruit(fruit));
  }

  render() {
    return (
      <div>
        <h3>Quick Add</h3>
        <button onClick={this.addFruitClick}>APPLE</button>
        <button onClick={this.addFruitClick}>ORANGE</button>
      </div>
    );
  }
}

export default FruitQuickAdd;
```

available in the fruit stand. The component also handles button clicks to dispatch an action to sell a fruit or to sell out all of the fruits.

Here's what the component looks like:

Listening for state changes and dispatching actions

Sometimes components need to listen for and render state updates *and* dispatch actions to the store. The `FruitSeller` component listens for state updates so that it can render a collection of buttons--one for each distinct fruit

```
// ./src/components/FruitSeller.js

import React from 'react';
import store from '../store';
import { sellFruit, sellOut } from '../actions/fruitActions';

class FruitSeller extends React.Component {
  sellFruitClick = (event) => {
    const fruit = event.target.innerText;
    store.dispatch(sellFruit(fruit));
  }

  sellOutClick = () => {
    store.dispatch(sellOut());
  }

  componentDidMount() {
    this.unsubscribe = store.subscribe(() => {
      this.forceUpdate();
    });
  }

  componentWillUnmount() {
    if (this.unsubscribe) {
      this.unsubscribe();
    }
  }

  render() {
    const fruit = store.getState();
    const distinctFruit = Array.from(new Set(fruit)).sort();

    if (distinctFruit.length === 0) {
      return null;
    }

    return (
      <div>
        <h3>Sell</h3>
        {distinctFruit.map((fruitName, index) => (
          <button key={index} onClick={this.sellFruitClick}>{fruitName}</button>
        ))}
      </div>
    );
  }
}
```

```
        <button onClick={this.sellOutClick}>SELL OUT</button>
      </div>
    );
  }
}

export default FruitSeller;
```

The `FruitSeller` component is sort of a mash up of the `FruitList` and `FruitQuickAdd` components!

Practicing on your own

There's just one component left to implement: `BulkAdd`. This is the perfect chance to get a bit of practice on your own to help you cement what you're learned in this article.

Reviewing a completed Fruit Stand example

To review and run a completed Fruit Stand example application that uses React with Redux, clone the [redux-fruit-stand-examples](#) repo.

After cloning the repo, open a terminal and browse to the `fruit-stand-redux-with-react` folder. Run the command `npm install` to install the project's dependencies. Then use the command `npm start` to run the Fruit Stand application.

This Fruit Stand example application is a React application created by the Create React App tooling. When running the application using `npm start`, the application should automatically open in your default browser. If it doesn't, you can manually browse to `http://localhost:3000/` to view the application.

What you learned

In this article, you learned how to add Redux actions, reducer(s), and a store to a React project. You also learned how to update a React class component to subscribe to a Redux store to listen for state changes and to dispatch actions to a Redux store.

Splitting and Combining Reducers

So far, you've been using a single reducer to manage state in your Redux store. As your applications increase in size and complexity, it'll become necessary to use multiple reducers, each managing a slice of state.

When you finish this article, you should be able to:

- Define multiple reducers to manage individual slices of state
- Use the Redux `combineReducers` method to combine multiple reducers into a single root reducer
- Update a reducer to delegate a state update to a subordinate reducer

Splitting reducers

Imagine that your fruit stand is extremely successful and it grows so much that you need multiple farmers helping you to keep your stand stocked with fruit. Your application's state will need to grow to store not only an array of `fruit` but also a `farmers` object that keeps track of your farmers.

Here's a sample state tree of your updated application:

```
{
  fruit: [
    'APPLE',
    'APPLE',
    'ORANGE',
    'GRAPEFRUIT',
    'WATERMELON',
  ],
  farmers: {
    1: {
      id: 1,
      name: 'John Smith',
      paid: false,
    },
    2: {
      id: 2,
      name: 'Sally Jones',
      paid: false,
    },
  },
}
```

The store now needs to handle new action types like `'HIRE_FARMER'` and `'PAY_FARMER'` by updating the `farmers` slice of state. You could add more cases to your existing reducer, but eventually the existing reducer would become too large and difficult to manage. The solution is to split the reducer into separate `fruit` and `farmers` reducers.

Splitting up the reducer into multiple reducers handling separate, independent *slices* of state is called **reducer composition**, and it's the fundamental pattern of building Redux apps. Because each reducer only handles a single *slice* of state, its `state` parameter corresponds only to the part of the state that it manages and it only responds to actions that concern that slice of state.

Split up your popular Fruit Stand application's reducer into two reducers:

- `fruitReducer` - A reducing function that handles actions updating the `fruits` slice of state
- `farmersReducer` - A reducing function that handles actions updating the new `farmers` slice of state

```
// ./src/reducers/fruitReducer.js

import {
  ADD_FRUIT,
  ADD_FRUITS,
  SELL_FRUIT,
  SELL_OUT,
} from '../actions/fruitActions';

const fruitReducer = (state = [], action) => {
  switch (action.type) {
    case ADD_FRUIT:
      return [...state, action.fruit];
    case ADD_FRUITS:
      return [...state, ...action.fruits];
    case SELL_FRUIT:
      const index = state.indexOf(action.fruit);
      if (index !== -1) {
        // remove first instance of action.fruit
        return [...state.slice(0, index), ...state.slice(index + 1)];
      }
      return state; // if action.fruit is not in state, return previous state
    case SELL_OUT:
      return [];
    default:
      return state;
  }
};

export default fruitReducer;
```

```
// ./src/reducers/farmersReducer.js

import { HIRE_FARMER, PAY_FARMER } from '../actions/farmersActions';

const farmersReducer = (state = {}, action) => {
  let nextState = Object.assign({}, state);
  switch (action.type) {
    case HIRE_FARMER:
      const farmerToHire = {
        id: action.id,
        name: action.name,
        paid: false
      };
      nextState[action.id] = farmerToHire;
      return nextState;
    case PAY_FARMER:
      const farmerToPay = nextState[action.id];
      farmerToPay.paid = !farmerToPay.paid;
      return nextState;
    default:
      return state;
  }
};

export default farmersReducer;
```

You'll also need to define a module containing the `'HIRE_FARMER'` and `'PAY_FARMER'` actions:

```
// ./src/actions/farmersActions.js

export const HIRE_FARMER = 'HIRE_FARMER';
export const PAY_FARMER = 'PAY_FARMER';

export const hireFarmer = (name) => ({
  type: HIRE_FARMER,
  id: new Date().getTime(),
  name,
});

export const payFarmer = (id) => ({
  type: PAY_FARMER,
  id,
});
```

Combining reducers

Your reducer setup is now much more modular. However, `createStore` only takes one `reducer` argument, so you must combine your reducers back into a single reducer to pass to the store. To do this you'll use the `combineReducers` method from the `redux` package and pass it an object that maps state keys to the reducers that handle those slices of state. Below, the `combineReducers` maps the `fruitReducer` for the `fruit` slice of state and the `farmersReducer` for the `farmers` slice of state. Invoking the `combineReducers` function returns a single `rootReducer` that you can use to create your Redux store.

```
// ./src/reducers/rootReducer.js

import { combineReducers } from 'redux';
import fruitReducer from './fruitReducer';
import farmersReducer from './farmersReducer';

const rootReducer = combineReducers({
  fruit: fruitReducer,
  farmers: farmersReducer
});

export default rootReducer;
```

```
import { createStore } from 'redux';
import rootReducer from './reducers/rootReducer';

const store = createStore(rootReducer);

export default store;
```

Delegating to reducers

Another aspect of reducer composition involves delegating state updates to subordinate reducers. Consider the farmers reducer. You can modify it so that the `farmers` (plural) reducer delegates to a `farmer` (singular) reducer whenever a single farmer's attributes need to be modified (in this case whenever a farmer has been hired or paid):

```
// ./src/reducers/farmersReducer.js

import { HIRE_FARMER, PAY_FARMER } from '../actions/farmersActions';

const farmerReducer = (state, action) => {
  // State is a farmer object.
  switch (action.type) {
    case HIRE_FARMER:
      return {
        id: action.id,
        name: action.name,
        paid: false
      };
    case PAY_FARMER:
      return Object.assign({}, state, {
        paid: !state.paid
      });
    default:
      return state;
  }
};

const farmersReducer = (state = {}, action) => {
  let nextState = Object.assign({}, state);
  switch (action.type) {
    case HIRE_FARMER:
      nextState[action.id] = farmerReducer(undefined, action);
      return nextState;
    case PAY_FARMER:
      nextState[action.id] = farmerReducer(nextState[action.id], action);
      return nextState;
    default:
      return state;
  }
};

export default farmersReducer;
```

Catching and preventing state mutation bugs

Updating the `farmersReducer` to delegate farmer state updates to the `farmerReducer` resolved a subtle state mutation bug. Take another look at the original implementation of the `farmersReducer` function:

```
const farmersReducer = (state = {}, action) => {
  let nextState = Object.assign({}, state);
  switch (action.type) {
    case HIRE_FARMER:
      const farmerToHire = {
        id: action.id,
        name: action.name,
        paid: false
      };
      nextState[action.id] = farmerToHire;
      return nextState;
    case PAY_FARMER:
      const farmerToPay = nextState[action.id];
      farmerToPay.paid = !farmerToPay.paid;
      return nextState;
    default:
      return state;
  }
};
```

Notice that the `state` parameter is duplicated to the `nextState` variable using the `Object.assign` method:

```
let nextState = Object.assign({}, state);
```

While this code correctly creates a duplicate of the `state` object, `nextState` is only a shallow duplicate as only the top-level object is duplicated. Each "farmer" object that the `state` object refers to are still the *same* objects.

In the `PAY_FARMER` case clause, the farmer object is mutated by setting the `paid` property to a new value:

```
case PAY_FARMER:
  const farmerToPay = nextState[action.id];
  farmerToPay.paid = !farmerToPay.paid;
  return nextState;
```

Now look again at the `PAY_FARMER` case clause in the version of the `farmersReducer` that delegates farmer state updates to the `farmerReducer`:

```
case PAY_FARMER:
  nextState[action.id] = farmerReducer(nextState[action.id], action);
  return nextState;
```

This code calls the `farmerReducer` by passing in the farmer object for the `action.id` property value (i.e. `nextState[action.id]`) and the `action` parameter. The `farmerReducer` has a `PAY_FARMER` case clause that correctly uses the `Object.assign` method to duplicate the farmer object with the new `paid` property value (i.e. `Object.assign({}, state, { paid: !state.paid })`):

```
const farmerReducer = (state, action) => {
  // State is a farmer object.
  switch (action.type) {
    case HIRE_FARMER:
      return {
        id: action.id,
        name: action.name,
        paid: false
      };
    case PAY_FARMER:
      return Object.assign({}, state, {
        paid: !state.paid
      });
    default:
      return state;
  }
};
```

Catching state mutation bugs is difficult to do. Leveraging patterns like reducer composition can help you from introducing these kinds of bugs in the first place.

Destructuring State in your component

If you try to start your Fruit Stand app now, you will probably get an error that looks something like:

```
TypeError: object is not iterable (cannot read property Symbol(Symbol.iterator))
```

That is because there is one last thing that you need to do in order prepare your fruitstand to use these split reducers: make sure that your component is accessing the right slice of state. Back in your `FruitList.js` `render` method, you are currently assigning the return value of your `getState()` call to 'fruit'.

```
const fruit = store.getState();
```

If you `console.log` or insert a debugger just after this line to see what fruit has been assigned, you will see:

```
fruit = {
  fruit: [],
  farmers: {}
}
```

Your state shape changed when you created these reducers! Your component is trying to iterate over your new state shape, instead of the fruit slice of that state. To give the component access to the array of fruit, destructure this assignment:

```
const { fruit } = store.getState();
```

Reviewing a completed Fruit Stand example

To review and run a completed Fruit Stand example application that contains multiple reducers, clone the [redux-fruit-stand-examples](#) repo.

After cloning the repo, open a terminal and browse to the `fruit-stand-redux-with-react-multiple-reducers` folder. Run the command `npm install` to install the project's dependencies. Then use the command `npm start` to run the Fruit Stand application.

This Fruit Stand example application is a React application created by the Create React App tooling. When running the application using `npm start`, the application should automatically open in your default browser. If it doesn't, you can manually browse to `http://localhost:3000/` to view the application.

What you learned

In this article, you learned how to define multiple reducers to manage individual slices of state. You also learned how to use the Redux `combineReducers` method to combine multiple reducers into a single root reducer and how to update a reducer to delegate a state update to a subordinate reducer.

Container Components

As you saw in an earlier article, there can be quite a bit of code involved in connecting a component to the store. Putting all this code into the component with heavy rendering logic tends to cause bloated components and violates the principle of [separation of concerns](#).

Therefore, it's a common pattern in Redux code to separate **presentational components** from their connected counterparts, called **containers**.

When you finish this article, you should be able to:

- Describe how container components differ from presentational components
- Write a container component to handle all of the Redux store interaction for one or more presentational components

Comparing presentational and container components

The distinction between presentational components and containers is not technical but rather functional. Presentational components are concerned with how things look and container components are concerned with how things work.

Here's a table outlining the differences:

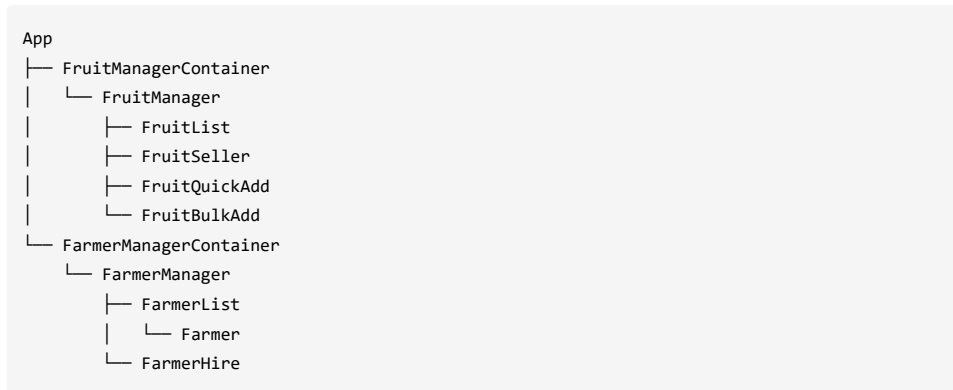
	Presentational	Container
Purpose	How things look (markup, styles)	How things work (data fetching, state updates)
Aware of Redux	No	Yes
To Read Data	Read data from <code>props</code>	Subscribe to Redux state
To Change Data	Invoke callbacks from <code>props</code>	Dispatch Redux actions

Note: You'll start with writing all of the code for your container components by hand. Later in this lesson, you'll learn how to create container components using the [React-Redux](#) library's `connect` method.

Determining where to create containers

Not every component needs to be connected to the store. Generally, you'll only want to create containers for the 'big' components in your app that represent sections of a page and contain smaller purely functional presentational components. These larger container components are responsible for interacting with the store and passing state and dispatch props down to all their presentational children.

For the Fruit Stand application, a good starting point would be to create two container components, `FruitManagerContainer` and `FarmerManagerContainer`, to respectively render the presentational components for the "Fruit" and "Farmers" sections of the page. Here's a visual representation of that component hierarchy:



Notice that the container component names are a combination of the name of the presentational component that they wrap and the suffix "Container".

In general, aim to have fewer containers rather than more. Most of the components you'll write will be presentational, but you'll need to generate a few containers to connect presentational components to the Redux store.

Writing a container component

While you can write a container component from scratch, you can also refactor an existing React component that interacts with a Redux store into separate container and presentational components.

Using a container component to retrieve state

Here's the current version of the `FruitList` component that subscribes to the store (using `store.subscribe`) to know when state has been updated and calls `store.getState` to retrieve and render the `fruit` state slice:

```
// ./src/components/FruitList.js

import React from 'react';
import store from '../store';

class FruitList extends React.Component {
  componentDidMount() {
    this.unsubscribe = store.subscribe(() => {
      this.forceUpdate();
    });
  }

  componentWillUnmount() {
    if (this.unsubscribe) {
      this.unsubscribe();
    }
  }

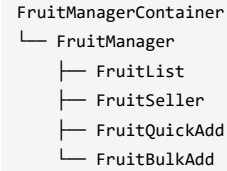
  render() {
    const { fruit } = store.getState();

    return (
      <div>
        {fruit.length > 0
          ? <ul>{fruit.map((fruitName, index) => <li key={index}>{fruitName}</li>)}
          : <span>No fruit currently in stock!</span>
        }
      </div>
    );
  }
}

export default FruitList;
```

The `FruitManager` component is responsible for rendering each of the fruit-related components (i.e. `FruitList`, `FruitSeller`, `FruitQuickAdd`, and `FruitBulkAdd`), so create a container component named `FruitManagerContainer` to handle all of the store interaction for the "Fruit" section of the page.

To review, here's what the component hierarchy will look like:



As a starting point, here's the code for the `FruitManagerContainer` component:

```
// ./src/components/FruitManagerContainer.js

import React from 'react';
import store from '../store';
import FruitManager from './FruitManager';

class FruitManagerContainer extends React.Component {
  componentDidMount() {
    this.unsubscribe = store.subscribe(() => {
      this.forceUpdate();
    });
  }

  componentWillUnmount() {
    if (this.unsubscribe) {
      this.unsubscribe();
    }
  }

  render() {
    const { fruit } = store.getState();

    return (
      <FruitManager fruit={fruit} />
    );
  }
}
```



```
export default FruitManagerContainer;
```

Notice that the container component, just like the original version of the `FruitList` component, subscribes to the store (using `store.subscribe`) to know when state has been updated and calls `store.getState` to retrieve the `fruit` state slice. But instead of directly rendering the `fruit` state, it sets a prop on the `FruitManager` component to pass the state down the component hierarchy.

The `FruitManager` component receives the `fruit` prop and in turn uses a prop to pass it down to the `FruitList` component:

```
// ./src/components/FruitManager.js

import React from 'react';
import FruitList from './FruitList';
import FruitSeller from './FruitSeller';
import FruitQuickAdd from './FruitQuickAdd';
import FruitBulkAdd from './FruitBulkAdd';

const FruitManager = ({ fruit }) => {
  return (
    <div>
      <h2>Available Fruit</h2>
      <FruitList fruit={fruit} />
      <h2>Fruit Inventory Manager</h2>
      <FruitSeller />
      <FruitQuickAdd />
      <FruitBulkAdd />
    </div>
  );
};

export default FruitManager;
```

And finally, the `FruitList` component receives the `fruit` prop and renders it into an unordered list:

```
// ./src/components/FruitList.js

import React from 'react';

const FruitList = ({ fruit }) => {
  return (
    <div>
      {fruit.length > 0
        ? <ul>{fruit.map((fruitName, index) => <li key={index}>{fruitName}</li>)}
          : <span>No fruit currently in stock!</span>
        }
    </div>
  );
};

export default FruitList;
```

Reminder: Using component props to pass a value down the component hierarchy is known as *prop threading*.

Notice that the `FruitList` *presentational* component, which no longer needs to use the `componentDidMount` and `componentWillUnmount` lifecycle methods to subscribe and unsubscribe to the store, can be refactored into a function component. Additionally, the `store` is no longer imported in the `FruitList` module, as the `FruitList` component simply receives and renders the `fruit` state via a prop without any knowledge of or direct interaction with the store.

Using a container component to dispatch actions

Here's the current version of the `FruitQuickAdd` component that dispatches the `ADD_FRUIT` action to add a fruit to the fruit stand:

```
// ./src/components/FruitQuickAdd.js

import React from 'react';
import store from '../store';
import { addFruit } from '../actions/fruitActions';

class FruitQuickAdd extends React.Component {
  addFruitClick = (event) => {
    const fruit = event.target.innerText;
    store.dispatch(addFruit(fruit));
  }

  render() {
    return (
      <div>
        <h3>Quick Add</h3>
        <button onClick={this.addFruitClick}>APPLE</button>
        <button onClick={this.addFruitClick}>ORANGE</button>
      </div>
    );
  }
}

export default FruitQuickAdd;
```

To prepare to refactor the `FruitQuickAdd` component into a *presentational* component, update the `FruitManagerContainer` component to the following code:

```
// ./src/components/FruitManagerContainer.js

import React from 'react';
import store from '../store';
import { addFruit } from '../actions/fruitActions';
import FruitManager from './FruitManager';

class FruitManagerContainer extends React.Component {
  componentDidMount() {
    this.unsubscribe = store.subscribe(() => {
      this.forceUpdate();
    });
  }

  componentWillUnmount() {
    if (this.unsubscribe) {
      this.unsubscribe();
    }
  }

  add = (fruit) => {
    store.dispatch(addFruit(fruit));
  }

  render() {
    const { fruit } = store.getState();

    return (
      <FruitManager
        fruit={fruit}
        add={this.add} />
    );
  }
}

export default FruitManagerContainer;
```

Notice that the `addFruit` action creator is imported (at the top of the file) and a new component method named `add` receives a `fruit` parameter value and calls `store.dispatch` to dispatch a `ADD_FRUIT` action. The `render` method

sets a prop on the `FruitManager` component to pass the `add` method down the component hierarchy.

The `FruitManager` component receives the `add` prop and in turn uses a prop to pass it down to the `FruitQuickAdd` component:

```
// ./src/components/FruitManager.js

import React from 'react';
import FruitList from './FruitList';
import FruitSeller from './FruitSeller';
import FruitQuickAdd from './FruitQuickAdd';
import FruitBulkAdd from './FruitBulkAdd';

const FruitManager = ({ fruit, add }) => {
  return (
    <div>
      <h2>Available Fruit</h2>
      <FruitList fruit={fruit} />
      <h2>Fruit Inventory Manager</h2>
      <FruitSeller />
      <FruitQuickAdd add={add} />
      <FruitBulkAdd />
    </div>
  );
};

export default FruitManager;
```

And finally, the `FruitQuickAdd` component receives the `add` callback function via a prop and calls it within a `handleClick` event handler, passing in the target button's inner text:

```
// ./src/components/FruitQuickAdd.js

import React from 'react';

const FruitQuickAdd = ({ add }) => {
  const handleClick = (event) => add(event.target.innerText);

  return (
    <div>
      <h3>Quick Add</h3>
      <button onClick={handleClick}>APPLE</button>
      <button onClick={handleClick}>ORANGE</button>
    </div>
  );
};

export default FruitQuickAdd;
```

The change between the original and refactored `FruitQuickAdd` component isn't as dramatic as the `FruitList` component example, but it's still a significant improvement to the overall separation of concerns. The `FruitQuickAdd` component is now strictly concerned with rendering the UI and handling user generated events (i.e. button clicks) and the `FruitManagerContainer` component is now strictly concerned with interacting with the Redux store.

Reviewing the completed container component

The `FruitManagerContainer` *container* component can continue to be expanded until each of its child *presentational* components no longer interact directly with the store. Here's a look at the completed `FruitManagerContainer` component:

```
// ../src/components/FruitManagerContainer.js

import React from 'react';
import store from '../store';
import { addFruit, addFruits, sellFruit, sellOut } from '../actions/fruitActions'
import FruitManager from './FruitManager';

class FruitManagerContainer extends React.Component {
  componentDidMount() {
    this.unsubscribe = store.subscribe(() => {
      this.forceUpdate();
    });
  }

  componentWillUnmount() {
    if (this.unsubscribe) {
      this.unsubscribe();
    }
  }

  add = (fruit) => {
    store.dispatch(addFruit(fruit));
  }

  addBulk = (fruit) => {
    store.dispatch(addFruits(fruit));
  }

  sell = (fruit) => {
    store.dispatch(sellFruit(fruit));
  }

  sellAll = () => {
    store.dispatch(sellOut());
  }

  render() {
    const { fruit } = store.getState();
    const distinctFruit = Array.from(new Set(fruit)).sort();

    return (
      <FruitManager
```

```
      fruit={fruit}
      distinctFruit={distinctFruit}
      add={this.add}
      addBulk={this.addBulk}
      sell={this.sell}
      sellAll={this.sellAll} />
    );
  }
}

export default FruitManagerContainer;
```

Reviewing a completed Fruit Stand example

To review and run a completed Fruit Stand example application that utilizes containers, clone the [redux-fruit-stand-examples](#) repo.

After cloning the repo, open a terminal and browse to the

`fruit-stand-redux-with-react-containers` folder. Run the command `npm install` to install the project's dependencies. Then use the command `npm start` to run the Fruit Stand application.

This Fruit Stand example application is a React application created by the Create React App tooling. When running the application using `npm start`, the application should automatically open in your default browser. If it doesn't, you can manually browse to `http://localhost:3000/` to view the application.

What you learned

In this article, you learned how container components differ from presentational components. You also learned how to write a container component to handle all of the Redux store interaction for one or more presentational components.

Freezing Objects

As you learned earlier in this lesson, a reducer must never mutate its arguments. If the state changes, the reducer must return a new object.

JavaScript provides us with an easy way to enforce this.

`Object.freeze` prevents new properties from being added to an object, and also prevents properties currently on an object from being altered or deleted. Essentially, it renders an object immutable, which is exactly what you want.

When you finish this article, you should be able to use `Object.freeze` to prevent the current state within a reducer from being mutated.

Using `Object.freeze` to prevent state mutations

By calling `Object.freeze(state)` at the top of every reducer, you can ensure that the state is never accidentally mutated. For example, this is what your farmer reducer from the Fruit Stand application would look like:

```
const farmersReducer = (state = {}, action) => {
  Object.freeze(state);
  let nextState = Object.assign({}, state);
  switch (action.type) {
    case HIRE_FARMER:
      const farmerToHire = {
        id: action.id,
        name: action.name,
        paid: false
      };
      nextState[action.id] = farmerToHire;
      return nextState;
    case PAY_FARMER:
      const farmerToPay = nextState[action.id];
      farmerToPay.paid = !farmerToPay.paid;
      return nextState;
    default:
      return state;
  }
};
```

Now you can be certain that you won't accidentally mutate the state within the reducer.

Understanding the difference between deep and shallow freezes

Here's another example:

```
const people = { farmers: { name: 'Old MacDonald' } };
Object.freeze(people);
```

When you try to mutate an object that you *froze* by modifying a property, it will be prevented:

```
people.farmers = { name: 'Young MacDonald' };
people; // { farmers: { name: 'Old MacDonald' } }
```

Note: This is not a *deep freeze*. `Object.freeze` performs a *shallow freeze* as it only applies to the immediate properties of the object itself. Nested objects can still be mutated, so be careful. Here's an example of this:

```
people.farmers.name = 'Young MacDonald';
people; // { farmers: { name: 'Young MacDonald' } }
```

`Object.freeze` and arrays

You can also use `Object.freeze` to freeze an array, so if a reducer's `state` parameter is an array, you can still prevent accidental state mutations:

```
const fruitReducer = (state = [], action) => {
  Object.freeze(state);
  switch (action.type) {
    case ADD_FRUIT:
      return [...state, action.fruit];
    case ADD_FRUITS:
      return [...state, ...action.fruits];
    case SELL_FRUIT:
      const index = state.indexOf(action.fruit);
      if (index !== -1) {
        // remove first instance of action.fruit
        return [...state.slice(0, index), ...state.slice(index + 1)];
      }
      return state; // if action.fruit is not in state, return previous state
    case SELL_OUT:
      return [];
    default:
      return state;
  }
};
```

When an array is frozen with `Object.freeze`, its elements cannot be altered and no elements can be added to or removed from the array. Just like with objects, freezing arrays has limitations. If the array's elements containing objects, properties on those objects can still be mutated.

What you learned

In this article, you learned how to use `Object.freeze` to prevent the current state within a reducer from being mutated.

Preloaded State

Currently, when the page in the browser is reloaded, any state data stored in the Redux store is lost. Later in this lesson, you'll learn how to use Redux to interact with an API to persist state data. Until then (or if your React application doesn't use an API), you can use the combination of Redux's ability to create a store with *preloaded state* with the browser's local storage to persist store state across page reloads.

When you finish this article, you should be able to:

- Create a Redux store with preloaded state
- Use the browser's local storage to persist a Redux store's state across page reloads

Creating a store with preloaded state

So far, your Redux stores have initialized with no initial state. Sometimes, though, you may want to take pre-existing data and pass it into the store upon initialization. Such data can be passed to the `createStore` method using the `preloadedState` argument:

```
const preloadedState = {
  fruit: [
    'APPLE',
    'ORANGE',
  ],
  farmers: {
    1: {
      id: 1,
      name: 'John Smith',
      paid: false,
    },
    2: {
      id: 2,
      name: 'Sally Jones',
      paid: false,
    },
  },
};

const store = createStore(rootReducer, preloadedState);
```

A couple of things to note about preloading state:

- The `preloadedState` must match the state shape (as produced by the reducers).
- `preloadedState` is not the same as default state. Default state should always be set in your reducers themselves.

Creating local storage helper functions

To assist with using the browser's local storage to persist a Redux store's state across page reloads, start with creating a set of helper functions:

```
// localStorage.js

const STATE_KEY = 'fruitstand';

export const loadState = () => {
  try {
    const stateJSON = localStorage.getItem(STATE_KEY);
    if (stateJSON === null) {
      return undefined;
    }
    return JSON.parse(stateJSON);
  } catch (err) {
    console.warn(err);
    return undefined;
  }
};

export const saveState = (state) => {
  try {
    const stateJSON = JSON.stringify(state);
    localStorage.setItem(STATE_KEY, stateJSON);
  } catch (err) {
    console.warn(err);
  }
};
```

The `saveState` function is responsible for converting the `state` parameter value into JSON (using the `JSON.stringify` method) and saving the state JSON string to the browser's local storage (using the `localStorage.setItem` method). A `try...catch` statement is used to catch and log any errors.

The `loadState` function is responsible for loading the state JSON from the browser's local storage (using the `localStorage.getItem` method). If the state JSON isn't stored in local storage, `undefined` is returned so the store's reducer function can initialize the state to its default value. If the state JSON is successfully retrieved from local storage, it's parsed into JavaScript objects (using the `JSON.parse` method) and returned to the caller. A `try...catch` statement is used to catch and log any errors.

Saving state to local storage

To ensure that the persisted state in local storage doesn't get out of sync with the store, you want to persist the state whenever it's updated. Knowing that the store's reducer is called whenever there's an action dispatched to update the state, you might be tempted to update your reducer like this:

```
import {
  ADD_FRUIT,
  ADD_FRUITS,
  SELL_FRUIT,
  SELL_OUT,
} from '../actions/fruitActions';
import { saveState } from '../localStorage';

const fruitReducer = (state = [], action) => {
  Object.freeze(state);
  switch (action.type) {
    case ADD_FRUIT:
      const nextState = [...state, action.fruit];
      saveState(nextState); // Persist state data to local storage
      return nextState;

    // Case clauses removed for brevity.

    default:
      return state;
  }
};

export default fruitReducer;
```

But don't do this! Per the [official Redux docs on reducers](#), reducers should stay *pure* and not cause *side effects* (like calling APIs or persisting data to local storage).

To keep your reducers pure, handle persisting state to local storage in the module where you create your store (`store.js`) by subscribing to listen for state changes:

```
import { createStore } from 'redux';
import rootReducer from '../reducers/rootReducer';
import { saveState } from '../localStorage';

const store = createStore(rootReducer);

store.subscribe(() => {
  saveState(store.getState());
});

export default store;
```

Now whenever the store's state is updated, the `store.getState` method is called to get and pass the current state to the `saveState` method.

Loading state from local storage

Now that you're persisting state to local storage, you can load state from local storage and pass it to the `createStore` method as preloaded state:

```
import { createStore } from 'redux';
import rootReducer from '../reducers/rootReducer';
import { loadState, saveState } from '../localStorage';

const preloadedState = loadState();

const store = createStore(rootReducer, preloadedState);

store.subscribe(() => {
  saveState(store.getState());
});
```



```
export default store;
```

With these updates in place, your Redux store's state will persist across page reloads.

What you learned

In this article, you learned how to create a Redux store with preloaded state. You also learned how to use the browser's local storage to persist a Redux store's state across page reloads.

React and Redux To-Do List Project

Today you'll be building a to-do list application with React and local storage. Instead of using Context to manage and update your application's state, you'll set up a Redux store and interact with it using the store's `getState`, `dispatch`, and `subscribe` methods.

This project will also give you a better understanding of how to share and update "global" data across a React application by using Redux. You'll use Redux to dispatch action POJOs through a reducer function, and have your component access an updated version of the Redux store's state.

In this project, you will:

- Generate a Redux store to manage your application's global information
- Define functions to save and load the Redux store's state with local storage
- Generate a Redux **store** with a preloaded state from local storage by using the `createStore` method from the Redux library

- Set up a **reducer** to direct different action types to interact with the Redux store in different ways
- Set up **actions** to create a task and delete a task
- Use a `debugger` to investigate the state from within a component

Phase 1: Set up project and Redux store

Begin by cloning the starter project from

<https://github.com/appacademy-starters/react-redux-todo-list-starter>.

Take a moment to examine the project's file tree below. In the next few phases, you'll follow the `TODO` notes in each file to implement Redux into your React project.

```
├─ package-lock.json
├─ package.json
├─ public
│   └─ index.html
└─ src
    ├─ App.js
    ├─ actions
    │   └─ taskActions.js
    ├─ components
    │   ├─ Task.js
    │   ├─ TodoForm.js
    │   └─ TodoList.js
    ├─ index.js
    ├─ localStorage.js
    ├─ reducers
    │   └─ tasksReducer.js
    └─ store.js
```

Local storage

Let's start by setting up some functions in the `localStorage.js` file to save and load the Redux store's state with local storage!

In the `loadState` function, you'll want to access the stored tasks state from local storage by using the `localStorage.getItem` method. If there is no state found, return `undefined`. However, if the state was found, parse the state from JSON into JavaScript and return the parsed state. If any errors were caught, log the errors with a `console.warn` statement and have the function return `undefined`.

In the `saveState` function, you'll want to parse the `state` input from JavaScript into a JSON string. When you call the `saveState` function, you'll invoke the function with the Redux store's state accessed with the `store.getState` method. After parsing the state from JavaScript into a JSON string, set the string into local storage. Lastly, you'll want to catch any errors with a `console.warn` statement.

Generate application's Redux store

Now that you've set up some functions to handle accessing and storing the data with local storage, you'll want to use those functions in the `store.js` file.

In this file, you'll use Redux's `createStore` function to set up your application's Redux `store`. As a reminder, the `createStore` function takes in a reducer as its first argument, and an optional preloaded state, also referred to as *initial* state, as its second argument.

Use the `loadState` function you just defined to access the `preloadedState`. Now you'll invoke the `createStore` function with the `tasksReducer` and the `preloadedState` to generate the application's Redux store.

You'll want your application to update local storage and log the state whenever there is an update to the store - this means you'll want your application to listen for changes to the store with the `store.subscribe` method and then update local storage with the `saveState` function and `console.log` the state upon any change.

Phase 2: Actions and reducers

Now that you have your application's Redux store set up, it's time to define some action creator functions and reducers! You'll define action creator functions in the `taskActions.js` file and set up corresponding case statements for each action type in the `tasksReducer.js` file.

Define action creator functions

As a reminder, it is best practice to use constants for action types, instead of string literals, to ensure that errors will be thrown for typos. Start by defining constants for your action types: `CREATE_TASK` and `DELETE_TASK`.

Once you have the constants set up, it's time to define an action creator function for each action type! Start by thinking of what payload information you want your action POJOs to pass into the reducer function.

Define a `createTask` action creator function that returns actions of type `CREATE_TASK`. You'll want `type`, `taskId`, and `taskMessage` payload keys for each `CREATE_TASK` action POJO. Have the action creator function take in a `taskMessage` and auto-generate the `taskId`. You can set the `taskId` to a time-string that is set when the action creator function is invoked. Generate a new `Date` object and get its time-string with `new Date().getTime()`. Set the time-string to the `taskId` payload key and the `taskMessage` input to the `taskMessage` payload key.

Now you'll want to define the `deleteTask` action creator function to return actions of type `DELETE_TASK`. You'll want the action creator function to take in a `taskId`. Each `DELETE_TASK` action POJO should have a `type` property and a `taskId` payload key.

Define tasks reducer function

The next step is to finish implementing the `tasksReducer` ! Begin by freezing the `state` with `Object.freeze(state)`; so that you won't accidentally mutate the state. As a reminder, Redux follows the immutable state pattern, meaning that a reducer function should never directly mutate state. After freezing the state, import `CREATE_TASK` and `DELETE_TASK` string literal constants and set up a switch statement to evaluate a case statement based on each `action.type` .

In the `CREATE_TASK` case, you'll want to make a copy of the state, structure a `newTask` POJO, and add the `newTask` into the copy of the state before returning the copy. Define a `nextState` variable and use spread syntax (`...`) to make a copy of the state (`{ ...state }`). Next, you'll want to structure the `newTask` POJO to have an `id` property set to the action's `taskId` payload and a `message` property set to the action's `taskMessage` payload.

Once you have finished structuring the `newTask` POJO, key into the `nextState` with the new task ID and set the value of `nextState[newTask.id]` to the `newTask` . Alternatively, you could use the `taskId` payload and set the value of `nextState[action.taskId]` to the `newTask` (this will also accomplish what we want, which is to set up a `nextState` with keys that are task IDs and values that are task POJOs). At the end of the `CREATE_TASK` case statement, return the updated `nextState` .

In the `DELETE_TASK` case, you'll also want to make a copy of the state (`{...state}`). Set the copy of the state to a `stateWithDeletion` variable. Since your `DELETE_TASK` actions have a `taskId` payload, you can use JavaScript's `delete` operator to delete a specific key-value pair from the `stateWithDeletion` object, based on the `taskId` payload:

```
delete stateWithDeletion[action.taskId];
```

The last thing left in your `DELETE_TASK` statement is to return the updated `stateWithDeletion` ! If you compare your initial definition of the `nextState` and `stateWithDeletion` variables, you'll see that they are both copies of the

`state` made with spread syntax. Move the `nextState` variable outside of the `switch` statement so that both `case` statements can reference and update the `nextState` , instead of the `DELETE_TASK` case statement creating a new copy of the state and updating it.

Phase 3: Dispatch actions from the DevTools console

Now you can test whether you can actually create a task by using the `store.dispatch` method to dispatch the `CREATE_TASK` action. As a reminder, dispatching the action will "send" it through the reducer to determine what operation to perform based on the action's `type` property. Take a moment to go into your `index.js` file and import your application's Redux `store` and action creator functions:

```
import { store } from './store';
import { createTask, deleteTask } from './actions/taskActions';
```

Now that you've had the store and actions imported into the file, you can set them as properties to the `window` object, so that you can access the `store` and actions from the developer tools console.

```
window.store = store;
window.createTask = createTask;
window.deleteTask = deleteTask;
```

At this point, your `index.js` file should look something like this:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import { store } from './store';
import { createTask, deleteTask } from './actions/taskActions';

window.store = store;
window.createTask = createTask;
window.deleteTask = deleteTask;

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Open up your browser's DevTools console and type `window.store`. Now you should see the `store` object and its methods:

```
{dispatch: f, subscribe: f, getState: f, replaceReducer: f, Symbol(observable): f}
,
```

Now type `window.store.getState()`. You should see an empty object - this is the default state (`state = {}`) that you set up in the `tasksReducer`.

Since you can access your application's state from the DevTools console, that means you can also dispatch actions by invoking the `window.store.dispatch` method with an action:

```
window.store.dispatch(window.createTask('learn redux'));
```

You just dispatched a `CREATE_TASK` action! You'll see that your updated state was logged - this is because of the `console.log` statement in the `store.subscribe` invocation in your `index.js` file (as you might remember, the `store.subscribe` method listens for any updates to the store, i.e. dispatch calls). Dispatch another `CREATE_TASK` action:

```
window.store.dispatch(window.createTask('learn react hooks'));
```

Now if you type `window.store.getState()` again, you'll see that the state return from the `store.getState` method is the same plain old JavaScript object as the state that was logged within the `store.subscribe` invocation.

Now let's place some `debugger` statements in the `tasksReducer` and `createTask` action creator function! Remember to make sure the `debugger` statement in your `tasksReducer` is **inside** a case statement. If the `debugger` is between the switch statement and a case statement, you will never hit that breakpoint!

```
const tasksReducer = (state = {}, action) => {
  Object.freeze(state);
  switch (action.type) {
    case CREATE_TASK:
      debugger;
  }
  // CODE SHORTENED FOR BREVITY
```

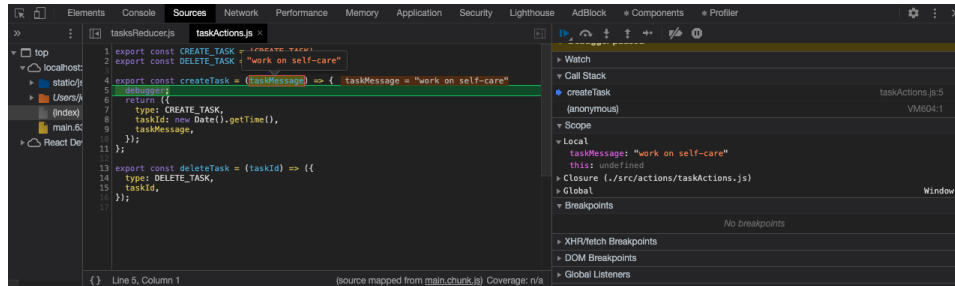
```
export const createTask = (taskMessage) => {
  debugger;
  return ({
    type: CREATE_TASK,
    taskId: new Date().getTime(),
    taskMessage,
  });
};
```

Now dispatch another `CREATE_TASK` action to hit the two `debugger` statements you just set:

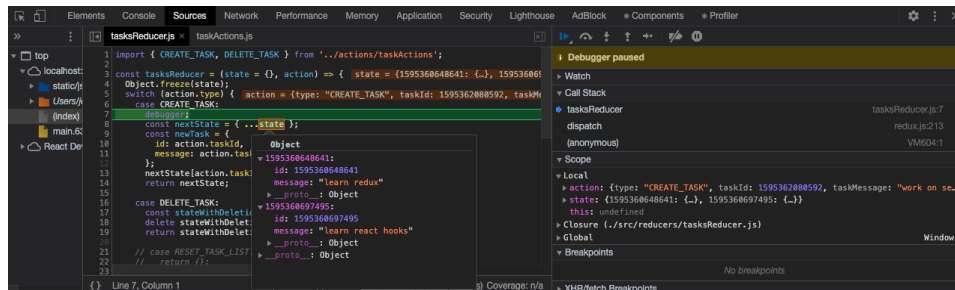
```
window.store.dispatch(window.createTask('work on self-care'));
```

Notice how you are now in the `Sources` tab of your DevTools looking at the `taskActions.js` file in your project. You can view the value of the

`taskMessage` argument by hovering over the variable or looking at the local scope variables in the DevTools' right window.



If you click the blue play button to continue to the next `debugger` statement, you'll land in your `tasksReducer.js` file and be able to hover over the `state` to view the value of the Redux store's previous state **before** the dispatching of the `CREATE_TASK` action.



Now if you click the blue play button to continue, you'll exit out of debug mode and your updated state will be logged in the console.

Congratulations! You just used a `debugger` to follow the Redux flow of dispatching a `CREATE_TASK` action! Comment out your `debugger` statements for now. In the next phase, you'll work on dispatching actions through a user interface.

Phase 4: Dispatch actions from components

Now it's time to set up a user interface that allows for intuitive dispatching of actions. In the `TodoForm` component, you'll set up a button that invokes the `createTask` action creator function with the `inputValue` state to dispatch a `CREATE_TASK` action based on the form input! For each `Task` component, you'll set up a button to dispatch a `DELETE_TASK` action for that task.

TodoForm

In the `TodoForm.js` file, import your application's Redux `store` instance and the `createTask` action creator function. Now you'll want to finish the `handleSubmit` method so that it dispatches a `CREATE_TASK` action. Invoke the `createTask` action creator function with the `inputValue` state and the `store.dispatch` method with the invoked action creator function.

Take a moment to test out the dispatch call generated by your form submission. Type a task in the input field - when you submit, you should see an updated state logged in the DevTools console with your new task!

TodoList

In the `TodoList.js` file, import the application's Redux `store` instance and the `deleteTask` action creator function. Now you'll set up the component's `componentDidMount` and `componentDidUnmount` life-cycle methods.

In the `componentDidMount` method, use the store's `subscribe` method to force a component to update whenever the state changes:

```
componentDidMount() {
  this.unsubscribe = store.subscribe(() => {
    this.forceUpdate();
  });
}
```

You want to name the subscription as `this.unsubscribe`, so that you can unsubscribe upon the unmounting of a component. When the `componentDidMount` life-cycle method is invoked upon the mounting of a component, it will invoke the `store.subscribe` method to force the component to update whenever the store's state changes. It will also set a `this.unsubscribe` variable to the `TodoList` class, so that `this.unsubscribe` is accessible from other parts of the component's code.

In the `componentDidUnmount` method, you'll want to check if the component has mounted by checking if `this.unsubscribe` has been defined. Whenever a component mounts, the `this.unsubscribe` variable set in the `componentDidMount` method will become initialize. If `this.unsubscribe` is undefined, that means that the component has not invoked the `componentDidMount` method and has therefore not been mounted yet. If `this.unsubscribe` is defined, you'll want to invoke `this.unsubscribe` to have the component unsubscribe from changes once component unmounts:

```
componentWillUnmount() {  
  if (this.unsubscribe) {  
    this.unsubscribe();  
  }  
}
```

In the `deleteTask` method, you'll want to wrap the invocation of the `deleteTask` action creator function with the `store.dispatch` method. The `deleteTask` action creator function will be invoked based on the `this.deleteTask` method's `id` input. Later in this phase, you'll pass the `TodoList` component's `this.deleteTask` method as a `deleteTask` prop into each `Task` component. Then, whenever the `deleteTask` prop is invoked from within a `Task` component, it can simply be invoked with a task ID to dispatch a `DELETE_TASK` action without needing to import the `store` into each `Task` component to invoke `store.dispatch`

In the component's `render` method, access the tasks stored in the Redux store's state by invoking the `store.getState` method and saving its return value to a `tasksState` variable. Now that you can use a `debugger` statement to view the state and check out what data you are working with!

If there are no tasks stored in state, you'll want to have the `TodoList` component return `null`. Otherwise if there are tasks stored in state, render a `Task` component for each of the tasks. For each `Task` component, you'll want to use the task's ID as the `key` and pass two props: the `task` object and the `this.deleteTask` method as a `deleteTask` prop.

Task

Have the `Task` function component destructure and take in the `deleteTask` method and `task` object props. Invoke the `deleteTask` function passed as a prop in the `Task` component's `handleClick` method and replace the `Hi, I'm a task in your to-do list!` placeholder text with the `task.message`.

As a reminder, the `deleteTask` action creator function was already wrapped with a `store.dispatch` call in the `TodoList` component - this is why the `handleClick` function in the `Task` component does not include a `store.dispatch` invocation. The `TodoList` component passed the wrapped function as a prop named `deleteTask` to each `Task` component. The `deleteTask` function invoked in the `Task` component's `handleClick` function is the `TodoList` component's `deleteTask` **method**, not the `deleteTask` **action creator function**.

Phase 5: Implement a full Redux cycle

In this phase, you'll implement a full Redux cycle without the guidance of `TODO` notes or specific, written instructions. Remember, the `debugger` statement is your friend! If you get stuck, think of where you can place

`debugger` statements to gain more context about your code. As a general guideline, feel free to follow the steps below:

- Set up an action creator function for a `RESET_TASK_LIST` action
- Set up a reducer case statement for the `RESET_TASK_LIST` action type
- Create a user interface (button) to dispatch the `RESET_TASK_LIST` action

Congratulations! You have just created an application that uses Redux to manage the application's information. Give yourself a pat on the back! As a reminder, the Redux library is a highly conceptual library to pick up, and when learning anything new practice always makes perfect! If the implementation of Redux feels confusing, always feel free to step back and use a `debugger` statement to follow the Redux flow: an action is generated, then the action is dispatched to go through a reducer, and then the store is updated.

WEEK-15 DAY-3

React + Redux

Redux Learning Objectives: Part 3

Just like Luke Skywalker returning to Dagobah to complete his Jedi training with Yoda, it's time for you to return to learning Redux one more time. In this final section on Redux, you'll learn how to use the React-Redux library to connect components to a Redux store and how to use middleware and thunks to interact with an API.

After completing this final section on Redux, you should be able to:

- Describe what a higher-order component (HOC) is
 - Write a higher-order component (HOC) that accepts a component as an argument and returns a new component
 - Use the React-Redux library's `<Provider />` component to make your Redux store available to any nested components that have been wrapped in the `connect` function
 - Use the React-Redux library's `connect` function to give a component access to a Redux store
 - Write a selector to extract and format information from state stored in a Redux store
 - Use the React-Redux library's `applyMiddleware` function to configure one or more middleware when creating a store
 - Write a thunk action creator to make an asynchronous request to an API and dispatch an action when the response is received
 - Describe a situation where defining multiple containers for a single component is advantageous
 - Configure a React application to use the Redux development tools
-

Higher-Order Components

In React, [higher-order components \(HOCs\)](#)

are a pattern for reusing component logic. Ultimately, higher order components allow you to dynamically generate wrapper components.

Before React existed, developers leveraged JavaScript's support for functions as first-class objects, meaning that functions can be treated just like any other data type (i.e. stored in a variable, object, or array, passed as an argument to a function, or returned from a function). Understanding what higher-order functions are and how they work will help you to understand and use higher-order components in React.

When you finish this article, you should be able to:

- Describe what a higher-order component (HOC) is
- Write a higher-order component (HOC) that accepts a component as an argument and returns a new component

Reviewing higher-order functions

Functions that operate on other functions, either by receiving them as arguments or returning them, are called **higher-order functions**. Said differently, higher-order functions are functions that:

- Define and return functions;
- Accept callbacks as arguments;
- Or do both.

Using closures with higher-order functions

A **closure**, also known as *lexical scoping*, is a function that uses **free variables**, variables defined outside of its scope. Closures come in handy when writing higher-order functions. Consider the following code:

```
const calculator = function (operationCb) { // high-order function
  return function (op1, op2) { // closure
    console.log(`calling with ${op1} ${op2}`);
    const result = operationCb(op1, op2);
    console.log(`equals ${result}`);
  };
}

const addition = function (n1, n2) { // callback
  console.log(`${n1} + ${n2}`);
  return n1 + n2;
}

const adder = calculator(addition);
adder(1, 2);
// calling with 1 2
// 1 + 2
// equals 3
```

The `calculator` function receives a callback as an argument (`operationCb`) which is called in the anonymous function `calculator` returns. This return value would not work if the inner function could not close over `operationCb`, a variable defined outside of its scope.

Defining higher-order functions with arrow functions

Arrow functions make it easy to write higher-order functions. The two examples below illustrate the same function:


```
// Without arrow functions (ES5):
function foo(arg1) {
  return function(arg2) {
    return function(arg3) {
      console.log(`${arg1} came before ${arg2} and ${arg3} came last`);
    };
  };
}

// With arrow functions (ES6):
const foo = arg1 => arg2 => arg3 => {
  console.log(`${arg1} came before ${arg2} and ${arg3} came last`);
};
```

Here's the earlier `calculator` function rewritten using arrow functions:

```
const calculator = (operationCb) => (op1, op2) => {
  console.log(`calling with ${op1} ${op2}`);
  const result = operationCb(op1, op2);
  console.log(`equals ${result}`);
};
```

Note: Remember, ES6 arrow functions, unlike normal JavaScript functions, are automatically bound to the context (`this`) that existed when they were defined. In other words, `this` means the same thing inside an arrow function that it does outside of it.

Leveraging higher-order components (HOCs)

In the same way that higher-order functions can receive a function as an argument and return a new function, **higher-order components** or **HOCs** receive a React component as an argument and return a new component.

ProtectedRoute and AuthRoute

In the **React Twitter Lite** project, you created two HOCs to control what pages users could see based upon their authentication status. The `ProtectedRoute` HOC ensured that only logged users could view the Profile and Home pages while the `AuthRoute` HOC prevented logged in users from viewing the Login or Registration pages:

```
export const ProtectedRoute = ({ component: Component, path, currentUserId, exact }) => {
  return (
    <Route
      path={path}
      exact={exact}
      render={(props) =>
        currentUserId ? <Component {...props} /> : <Redirect to="/login" />
      }
    />
  );
};
```

```
export const AuthRoute = ({ component: Component, path, currentUserId, exact }) => {
  return (
    <Route
      path={path}
      exact={exact}
      render={(props) =>
        currentUserId ? <Redirect to="/" /> : <Component {...props} />
      }
    />
  );
};
```

Notice how both components accept a component via the `component` parameter and return a `Route` component (that renders the passed component via a `render` prop).

Using HOCs to keep code DRY

HOCs give React developers a powerful way to reuse component logic so that they can keep their code DRY.

When learning about Redux container components, you might have noticed that the `FruitManagerContainer` and `FarmerManagerContainer` components in the Fruit Stand application contain the same `componentDidMount` and `componentWillUnmount` lifecycle method implementations:

```
// ../src/components/FruitManagerContainer.js

import React from 'react';
import store from '../store';
import {
  addFruit,
  addFruits,
  sellFruit,
  sellOut,
} from '../actions/fruitActions';
import FruitManager from './FruitManager';

class FruitManagerContainer extends React.Component {
  componentDidMount() {
    this.unsubscribe = store.subscribe(() => {
      this.forceUpdate();
    });
  }

  componentWillUnmount() {
    if (this.unsubscribe) {
      this.unsubscribe();
    }
  }

  add = (fruit) => {
    store.dispatch(addFruit(fruit));
  }

  addBulk = (fruit) => {
    store.dispatch(addFruits(fruit));
  }

  sell = (fruit) => {
    store.dispatch(sellFruit(fruit));
  }

  sellAll = () => {
    store.dispatch(sellOut());
  }

  render() {
    return (
      <FruitManager
        add={this.add}
        addBulk={this.addBulk}
        sell={this.sell}
        sellAll={this.sellAll}
      />
    );
  }
}
```

```

const { fruit } = store.getState();
const distinctFruit = Array.from(new Set(fruit)).sort();

return (
  <FruitManager
    fruit={fruit}
    distinctFruit={distinctFruit}
    add={this.add}
    addBulk={this.addBulk}
    sell={this.sell}
    sellAll={this.sellAll} />
);
}
}

export default FruitManagerContainer;

```

```

// ./src/components/FarmerManagerContainer.js

import React from 'react';
import store from '../store';
import { hireFarmer, payFarmer } from '../actions/farmersActions';
import FarmerManager from './FarmerManager';

class FarmerManagerContainer extends React.Component {
  componentDidMount() {
    this.unsubscribe = store.subscribe(() => {
      this.forceUpdate();
    });
  }

  componentWillUnmount() {
    if (this.unsubscribe) {
      this.unsubscribe();
    }
  }

  pay = (id) => {
    store.dispatch(payFarmer(id));
  }

  hire = (name) => {
    store.dispatch(hireFarmer(name));
  }

  render() {
    const { farmers: farmersState } = store.getState();
    const farmers = Object.values(farmersState);

    return (
      <FarmerManager
        farmers={farmers}
        pay={this.pay}
        hire={this.hire} />
    );
  }
}

export default FarmerManagerContainer;

```

While the amount of duplicated code is currently relatively small, keep in mind that **the approach of using the `forceUpdate` method to render the component whenever state is updated in the store isn't optimal**. Calling `forceUpdate` causes `render` to be called without first calling `shouldComponentUpdate`.

Ideally, each container component should contain logic to determine if the state that it retrieves from the store has *actually changed* before continuing with rendering. Adding the code for this logic would increase the amount of code duplicated in each container component.

React-Redux `connect`

[React-Redux](#), a library from the creators of [Redux](#), includes a higher-order component named `connect` that you can use to create container components. Using `connect` frees you from having to manually create the `FruitManagerContainer` and `FarmerManagerContainer` container components. Using `connect` also eliminates boilerplate code so you can focus on what makes each container unique: selecting specific slices of state from the store and writing functions to dispatch specific actions. Even better, `connect` includes the logic that's needed to optimize the rendering performance, only rendering when the state retrieved from the store has *actually changed*.

Over the next couple of articles, you'll learn how to use `connect` and the [React-Redux](#) library.

Writing a HOC for creating container components

Higher-order components like [React-Redux](#) `connect`, while extremely useful, can be confusing to understand. To help give you some insight into the underlying implementation of the `connect` function, you can build a your own *custom version* of `connect` (albeit without the logic that's needed to optimize rendering performance).

Don't worry if you struggle to fully understand the following custom version of `connect`. It takes time and practice to get comfortable with the techniques that are used to create higher-order components. And remember, **you don't have to build your `connect` function!** Going forward, you'll use the well-maintained, highly optimized `connect` function provided by the [React-Redux](#) library.

Consider the following *custom* `connect` function:

```
// ./src/connect.js

import React from 'react';
import store from './store';

const connect = (mapStateToProps, mapDispatchToProps) => (WrappedComponent) => (
  class extends React.Component {
    render() {
      let stateProps = {};
      if (mapStateToProps) {
        stateProps = mapStateToProps(store.getState(), this.props);
      }

      let dispatchProps = {};
      if (mapDispatchToProps) {
        dispatchProps = mapDispatchToProps(store.dispatch, this.props);
      }

      const propsToSpread = Object.assign({}, this.props, stateProps, dispatchProps);

      return (
        <WrappedComponent {...propsToSpread} />
      );
    }

    componentDidMount() {
      this.unsubscribe = store.subscribe(() => {
        this.forceUpdate();
      });
    }

    componentWillUnmount() {
      if (this.unsubscribe) {
        this.unsubscribe();
      }
    }
  }
);

export default connect;
```

The `connect` function is a higher-order component that returns a function. That function returns an anonymous class component that wraps the component passed in via the `WrappedComponent` parameter.

The anonymous class component is a "generic" container component that *connects* `WrappedComponent` to the Redux store by:

- Defining the `componentDidMount` and `componentWillUnmount` lifecycle method implementations necessary for managing the subscription to the store to render `WrappedComponent` when state is updated; and
- Passing slices of state and functions to dispatch actions down to `WrappedComponent` using props.

Comparing nested arrow functions to regular functions

As you saw earlier in this article, higher-order functions can be written using arrow functions or regular functions. The `connect` higher-order function can be written using arrow functions and implicit return statements:

```
const connect = (mapStateToProps, mapDispatchToProps) => (WrappedComponent) => (
  class extends React.Component {
    // Code removed for brevity.
  }
);
```

Or can be rewritten using regular functions and explicit return statements:

```
function connect(mapStateToProps, mapDispatchToProps) {
  return function(WrappedComponent) {
    return class extends React.Component {
      // Code removed for brevity.
    };
  };
};
```

The arrow function syntax, while concise, can be confusing to read.

Dynamically setting component attributes

The `connect` function's `mapStateToProps` and `mapDispatchToProps` parameters are both functions that are called within the anonymous class' `render` method:

```
const connect = (mapStateToProps, mapDispatchToProps) => (WrappedComponent) => (
  class extends React.Component {
    render() {
      let stateProps = {};
      if (mapStateToProps) {
        stateProps = mapStateToProps(store.getState(), this.props);
      }

      let dispatchProps = {};
      if (mapDispatchToProps) {
        dispatchProps = mapDispatchToProps(store.dispatch, this.props);
      }

      // Code removed for brevity.
    }

    // Code removed for brevity.
  }
);
```

The Redux store's current state, retrieved using `store.getState`, is passed to the `mapStateToProps` function and the store's `dispatch` method is passed to

the `mapDispatchToProps` function. The `mapStateToProps` and `mapDispatchToProps` functions also receive the anonymous class component's props (via `this.props`).

But what are the `mapStateToProps` and `mapDispatchToProps` functions exactly? To answer that question, take a look at an example of using the `connect` function to define the `FruitManagerContainer` component:

```
// ./src/components/FruitManagerContainer.js

import connect from '../connect';
import {
  addFruit,
  addFruits,
  sellFruit,
  sellOut,
} from '../actions/fruitActions';
import FruitManager from '../FruitManager';

const mapStateToProps = (state) => ({
  fruit: state.fruit,
  distinctFruit: Array.from(new Set(state.fruit)).sort(),
});

const mapDispatchToProps = (dispatch) => ({
  add: (fruit) => dispatch(addFruit(fruit)),
  addBulk: (fruit) => dispatch(addFruits(fruit)),
  sell: (fruit) => dispatch(sellFruit(fruit)),
  sellAll: () => dispatch(sellOut()),
});

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(FruitManager);
```

`mapStateToProps` is a function that accepts store's current state and returns an object that maps state data to one or more properties. `mapDispatchToProps` is also a function, but it accepts the store's `dispatch` method and returns an

object whose properties are set to functions that can be called dispatch actions to the store.

In the `connect` function, `Object.assign` is then used to combine the `stateProps` and `dispatchProps` objects with `this.props` (the anonymous class component's props):

```
const propsToSpread = Object.assign({}, this.props, stateProps, dispatchProps);
```

Then the properties defined on the `propsToSpread` object become props on `WrappedComponent` using JSX spread attribute syntax (`...`):

```
return (  
  <WrappedComponent {...propsToSpread} />  
);
```

Within the `FruitManager` component (the component that the `connect` function's `WrappedComponent` is wrapping), parameter destructuring is used to reference the props provided by the `mapStateToProps` and `mapDispatchToProps` functions. These props are then passed as props to the appropriate child presentational component:

```
// ./src/components/FruitManager.js  
  
import React from 'react';  
import FruitList from './FruitList';  
import FruitSeller from './FruitSeller';  
import FruitQuickAdd from './FruitQuickAdd';  
import FruitBulkAdd from './FruitBulkAdd';  
  
const FruitManager = ({ fruit, distinctFruit, add, addBulk, sell, sellAll }) => {  
  return (  
    <div>  
      <h2>Available Fruit</h2>  
      <FruitList fruit={fruit} />  
      <h2>Fruit Inventory Manager</h2>  
      <FruitSeller distinctFruit={distinctFruit} sell={sell} sellAll={sellAll} />  
      <FruitQuickAdd add={add} />  
      <FruitBulkAdd addBulk={addBulk} />  
    </div>  
  );  
};  
  
export default FruitManager;
```

Visualizing the flow of data all the from the `mapStateToProps` and `mapDispatchToProps` function definitions within the `FruitManagerContainer.js` file through the `connect` higher-order component down to the `FruitManager` wrapped component is **difficult to do**. Understanding how to use React's ability to dynamically set component attributes using JSX spread attribute syntax is **one of the more challenging aspects of writing higher-order components**.

Just remember that you don't need to fully understand the above example to use the `connect` function provided by the [React-Redux](#) library.

Reviewing a completed Fruit Stand example

To review and run a completed Fruit Stand example application that utilizes the above `custom connect` higher-order component to create container components, clone the [redux-fruit-stand-examples](#) repo.

After cloning the repo, open a terminal and browse to the `fruit-stand-redux-with-react-generic-container` folder. Run the command `npm install` to install the project's dependencies. Then use the command `npm start` to run the Fruit Stand application.

This Fruit Stand example application is a React application created by the Create React App tooling. When running the application using `npm start`, the application should automatically open in your default browser. If it doesn't, you can manually browse to `http://localhost:3000/` to view the application.

What you learned

In this article, you learned about higher-order components (HOCs) including how to write a higher-order component that accepts a component as an argument and returns a new component.

React-Redux: `<Provider/>`

As you learned in earlier articles, the integration techniques that you were initially shown were just a starting point with using Redux with React. Now that you've learned the basics of how React components interact with a Redux store, it's time to learn how you can use [React-Redux](#), a library from the creators of [Redux](#), to improve upon those techniques.

To prepare to use the `connect` function from the [React-Redux](#) library, you need to first add a `<Provider />` component to your

React application. When you finish this article, you should be able to use the `<Provider />` component to make your Redux store available to any nested components that have been wrapped in the `connect` function.

Understanding the advantages of the `<Provider />` component

Oftentimes, a deeply nested component will need access to the store, while its parents do not. Using vanilla React, these parents would have to receive the `store` prop in order to pass it down to its child.

Consider the example below:

```
// App.js

import React from 'react';

const UserInfo = ({ store }) => (
  <div>
    {store.getState().username}
  </div>
);

const Header = ({ store }) => (
  <div>
    <UserInfo store={store} />
  </div>
);

const App = ({ store }) => (
  <div>
    <Header store={store} />
  </div>
);

export default App;
```



```
// index.js

import React from 'react';
import ReactDOM from 'react-dom';

import { createStore } from 'redux';
import reducer from './reducer';
import App from './App';

const store = createStore(reducer);

ReactDOM.render(
  <React.StrictMode>
    <App store={store} />
  </React.StrictMode>,
  document.getElementById('root')
);
```

The `store` is created in the `index.js` file, but the `UserInfo` component that needs to access it is deeply nested. Thus, the store must be passed as a prop down the entire component tree, even though components such as the `Header` do not need to access the store.

This pattern, called **prop threading**, is tedious and error-prone. You can avoid it by using the `<Provider />` / `connect` API provided by React-Redux.

Preparing your React application for server-side rendering

Using `<Provider />` also helps to prepare your React/Redux application to utilize server-side rendering. Server-side rendering allows you to render components to static markup, which can help to reduce the initial loading time of your application.

React server-side rendering is an advanced topic that won't be covered in this course. For more information, see the official [React](#) and [Redux](#) docs.

Adding `<Provider />`

Before adding `<Provider />` to your React application, use npm to install the `react-redux` package:

```
npm install react-redux
```

Then, in the entry point for your application (typically the `index.js` file), import the `Provider` component and your Redux `store`:

```
import { Provider } from 'react-redux';
import store from './store';
```

Then use the `Provider` component to wrap your `App` component and set its `store` prop to your Redux `store`:

```
<Provider store={store}>
  <App />
</Provider>
```

Here's what your completed `index.js` file will look like:

```
// ./src/index.js

import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import './index.css';
import App from './App';
import store from './store';

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
);
```

`<Provider />` is simply a React component in which you wrap the rest of the application. The `Provider` component receives the `store` as a `prop` and sets a store `context`. Because you wrapped the entire `App` in the `Provider` component, all your components can access the store `context`.

Components that need to access the store `context` have to be wrapped in a container component created by the `connect` function, which converts the store `context` into a `store` prop. You'll learn how to use the `connect` function in the next article.

Understanding how `<Provider />` relates to the React Context API

The store `context` set by `<Provider />` is the *same* React Context that you used in an earlier lesson to manage global state within a React application. You can see this in action by reviewing the `react-redux` source code on GitHub:

```
// https://github.com/reduxjs/react-redux/blob/master/src/components/Context.js

import React from 'react'

export const ReactReduxContext = /*#__PURE__*/ React.createContext(null)

if (process.env.NODE_ENV !== 'production') {
  ReactReduxContext.displayName = 'ReactRedux'
}

export default ReactReduxContext
```

And while it's rarely used, it's possible to import the context from React-Redux and use the `<Consumer />` to access the `store`:

```
import { ReactReduxContext } from 'react-redux';

// in your connected component
render() {
  return (
    <ReactReduxContext.Consumer>
      {{{ store }}} => {
        // do something with the store here
      }
    </ReactReduxContext.Consumer>
  );
}
```

You can also connect the Redux `<Provider />` component to the `<Context.Provider />` component that passes the `value` of a context object to all child components. Redux's `<Provider />` component simply passes the Redux `store`, instead of a context `value`.

What you learned

In this article, you learned how to use the `<Provider />` component to make your Redux store available to any nested components that have been wrapped in the `connect` function.

React-Redux: `connect()`

The React-Redux library allows you to access the store `context` set by the `<Provider />` in a powerful and convenient way via the `connect` function. Using `connect`, you can pass specific slices of the store's state and specific action-dispatches to a React component as `props`. A component's `props` then serve as its API to the store, making the component more modular and less burdened by Redux boilerplate.

When you finish this article, you should be able to use the `connect` function to give a component access to a Redux store.

Calling `connect`

The React-Redux `connect` function is a *higher-order function*. It takes two arguments (plus a couple optional arguments you can read more about in the [docs](#)) and returns a function:

```
const createConnectedComponent = connect(  
  mapStateToProps,  
  mapDispatchToProps  
);
```

The returned function (`createConnectedComponent`) then takes the React component that needs access to the Redux store and returns a new React component:

```
const ConnectedComponent = createConnectedComponent(MyComponent);  
  
export default ConnectedComponent;
```

`ConnectedComponent` will render `MyComponent`, passing along `props` as determined by the `mapStateToProps` and `mapDispatchToProps` arguments.

You can combine these function calls into a single statement by immediately calling the function returned by the `connect` method (similarly to how you immediately call a function expression when defining an [IIFE](#)):

```
const ConnectedComponent = connect(mapStateToProps, mapDispatchToProps)(MyComponent);  
  
export default ConnectedComponent;
```

Typically, to keep things as concise as possible, the `ConnectedComponent` variable is omitted:

```
export default connect(mapStateToProps, mapDispatchToProps)(MyComponent);
```

Defining `mapStateToProps(state, [ownProps])`

This first argument to `connect` is a function, `mapStateToProps`. It tells `connect` how to map the `state` into your component's `props`.

It must take as an argument the store's `state` (supplied by the `Provider`'s store `context`) and return an object containing the relevant `props` for your component.

```
const MyComponent = ({ name }) => (
  <div>{name}</div>
);

const mapStateToProps = (state) => ({
  name: state.name;
});

const ConnectedComponent = connect(mapStateToProps)(MyComponent);
```

In the example above, `ConnectedComponent` will render `MyComponent`, passing `name` as a prop.

ownProps (optional)

A component with explicit `props` passed down from its parent (e.g. `<ConnectedComponent lastName='Wozniak' />`) can merge those `props` with slices of `state` via `ownProps`, a optional second argument to `mapStateToProps`:

```
const mapStateToProps = (state, ownProps) => ({
  firstName: state.name,
  initials: `${state.name[0]}. ${ownProps.lastName[0]}.`
});

const ConnectedComponent = connect(mapStateToProps)(MyComponent);
```

You can also access React Router props, such `match` and `history` through `ownProps`. Imagine you have a `users` slice of state, and you want to pass a specific user's `name` based on a `:userId` parameter. You can access the parameter from within the `mapStateToProps` function with `ownProps.match.params.userId`:

```
const mapStateToProps = (state, ownProps) => ({
  name: state.users[ownProps.match.params.userId].name,
});

const ConnectedComponent = connect(mapStateToProps)(MyComponent);
```

Defining mapDispatchToProps

`mapDispatchToProps` is the second argument to `connect`. It's a function that accepts the store's `dispatch` method and returns an object containing functions that can be called to dispatch actions to the store. These action dispatchers are then passed as `props` to the component.

```
const deleteTodo = (id) => ({ type: 'DELETE_TODO', id }); // action creators
const addTodo = (msg) => ({ type: 'ADD_TODO', msg });

const mapDispatchToProps = (dispatch) => ({
  handleDelete: (id) => dispatch(deleteTodo(id)),
  handleAdd: (msg) => dispatch(addTodo(msg))
});

const ConnectedComponent = connect(null, mapDispatchToProps)(MyComponent);
```

Notice that in the example above, the `connect` function is invoked with `null` as a placeholder for the `mapStateToProps` function. The `connect` function expects `mapStateToProps` as its first argument and `mapDispatchToProps` as its second argument.

Putting it all together

```
const MyComponent = ({ firstName, initials, handleAdd, handleDelete }) => {
  return <div>...</div>;
};

const mapStateToProps = (state, ownProps) => ({
  firstName: state.name,
  initials: `${state.name[0]}. ${ownProps.lastName[0]}.`
});

const mapDispatchToProps = (dispatch) => ({
  handleDelete: (id) => dispatch(deleteTodo(id)),
  handleAdd: (msg) => dispatch(addTodo(msg))
});

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(MyComponent);
```

`MyComponent` will receive `firstName`, `initials`, `handleDelete`, and `handleAdd` as `props`.

And remember, unlike the earlier attempt at writing a custom `connect` higher-order component, the React-Redux library's `connect` function **contains logic to optimize the rendering of your connected components**.

Reviewing a completed Fruit Stand example

To review and run a completed Fruit Stand example application that utilizes the React-Redux library, clone the [redux-fruit-stand-examples](#) repo.

After cloning the repo, open a terminal and browse to the `fruit-stand-redux-with-react-official-bindings` folder. Run the command `npm install` to install the project's dependencies. Then use the command `npm start` to run the Fruit Stand application.

This Fruit Stand example application is a React application created by the Create React App tooling. When running the application using `npm start`, the application should automatically open in your default browser. If it doesn't, you can manually browse to `http://localhost:3000/` to view the application.

What you learned

In this article, you learned how to use the React-Redux library's `connect` function to give a component access to a Redux store.

Selectors

Selectors are functions used to extract and format information from the application state in different forms. When you finish this article, you should be able to write a selector to extract and format information from state stored in a Redux store.

Writing a selector

Here's a sample state tree from the Fruit Stand React/Redux application:

```
{
  fruit: [
    'APPLE',
    'APPLE',
    'ORANGE',
    'GRAPEFRUIT',
    'WATERMELON',
  ],
  farmers: {
    1: {
      id: 1,
      name: 'John Smith',
      paid: false,
    },
    2: {
      id: 2,
      name: 'Sally Jones',
      paid: false,
    },
  }
}
```

The state's `farmers` are stored as an object. Keys correspond to `farmer.id`s and values correspond to `farmer` objects. This yields $O(1)$ for the lookup of a single farmer. However, storing all the farmers as values of an object makes it slightly inconvenient to obtain and render them all at once. To solve this inconvenience, we use selectors.

Selectors are typically defined in a file that sits next to the reducer for its slice of state. For example, if the `farmers` state slice is managed by the reducer defined in `./src/reducers/farmersReducer.js`, then the farmers selectors would be stored in a file at `./src/reducers/farmersSelectors.js`.

Selectors are passed the application's `state` and return information from the state in a specified form (e.g. an array). You can use selectors to format different slice(s) of the state by calling them in a container's `mapStateToProps`.

For example, `getAllFarmers` returns all the farmers stored in the state as an array of `farmer` objects, making it easier to iterate over and render each one.

```
// ./src/reducers/farmersSelectors.js

export const getAllFarmers = ({ farmers }) => (
  Object.values(farmers)
);
```

A selector can be used in multiple components' `mapStateToProps`. For example:

```
// ./src/components/FarmerManagerContainer.js

import { getAllFarmers } from '../reducers/farmersSelectors';

const mapStateToProps = (state) => ({
  farmers: getAllFarmers(state),
});
```

Selectors are passed the entire application `state` so they can utilize multiple slices of the application state to assemble data. For example, if the Fruit Stand application's state tree included a `filter` state slice:

```
{
  fruit: [
    'APPLE',
    'APPLE',
    'ORANGE',
    'GRAPEFRUIT',
    'WATERMELON',
  ],
  farmers: {
    1: {
      id: 1,
      name: 'John Smith',
      paid: false,
    },
    2: {
      id: 2,
      name: 'Sally Jones',
      paid: false,
    },
  },
  filter: ''
}
```

Then you could write a selector to extract a filtered list of `farmer` objects:

```
// ./src/reducers/farmersSelectors.js

export const getAllFarmers = ({ farmers }) => (
  Object.values(farmers)
);

export const getFilteredFarmers = ({ farmers, filter }) => {
  const lowerCaseFilter = filter.toLowerCase();
  return Object.values(farmers).filter(
    (farmer) => farmer.name.toLowerCase().includes(lowerCaseFilter)
  );
};
```

```
// ./src/components/FarmerManagerContainer.js

import { getAllFarmers, getFilteredFarmers } from '../reducers/farmersSelectors';

const mapStateToProps = (state) => ({
  farmers: getAllFarmers(state),
  filteredFarmers: getFilteredFarmers(state),
});
```

Selector examples

```
// ./src/reducers/farmersSelectors.js

// Returns the state's farmers as an array of farmer objects.
export const getAllFarmers = ({ farmers }) => (
  Object.values(farmers)
);

// Returns the state's farmers as an array of farmer objects,
// filtered by their name.
export const getFilteredFarmers = ({ farmers, filter }) => {
  const lowerCaseFilter = filter.toLowerCase();
  return Object.values(farmers).filter(
    (farmer) => farmer.name.toLowerCase().includes(lowerCaseFilter)
  );
};

// Returns the selected farmer object or an empty farmer object
// if no farmer exists with given id.
export const selectFarmer = ({ farmers }, id) => {
  const nullFarmer = {
    id: null,
    name: '',
    paid: false
  };
  return farmers[id] || nullFarmer;
};
```

What you learned

In this article, you learned how to write a selector to extract and format information from state stored in a Redux store.

Middleware

In Redux, middleware specifically refers to an `enhancer` passed to the store via `createStore`. When a `dispatch` is made, the middleware intercepts the `action` before it reaches the `reducer`. The middleware can then:

- **resolve the action itself** (for example, by making an AJAX request),
- **pass along the action** (if the middleware isn't concerned with it),
- **generate a side effect** (such as logging debugging information),
- **send another dispatch** (if the action triggers other actions),
- or some combination thereof.

You'll use Redux middleware for logging information about the store and making asynchronous API requests, but you can also use it for crash reporting, routing, and many other applications.

When you finish this article, you should be able to use the React-Redux library's `applyMiddleware` function to configure one or more middleware when creating a store.

Applying middleware to a Redux store

Recall the `redux` library's `createStore` function used to instantiate a store. `createStore` accepts three arguments (`reducer`, `preloadedState`, `enhancer`); middleware is given to the store via the optional `enhancer` argument.

Consider the following example, where you import a third-party `logger` middleware:

```
// ./src/store.js

import { createStore, applyMiddleware } from 'redux';
import logger from 'redux-logger';

import rootReducer from './reducers/rootReducer';

const configureStore = (preloadedState = {}) => {
  return createStore(
    rootReducer,
    preloadedState,
    applyMiddleware(logger),
  );
};

export default configureStore;
```

Any actions dispatched to the `store` pass through the `logger` middleware, which prints the store's state before and after the `action` is processed.

Note: `applyMiddleware()` accepts multiple arguments, so you can also apply more middleware if necessary.

Reviewing the signature of middleware functions

In addition to importing third-party middlewares such as the above `logger`, you'll sometimes need to roll your own. All middleware functions need to conform to the same signature in order to be compatible with the store and other middlewares.

A **function signature** is the set of inputs and output of a function. A Redux middleware must always have the following signature:


```
const middleware = store => next => action => {
  // side effects, if any
  return next(action);
};
```

Every middleware receives the `store` as an argument and returns a function that takes the `next` link in the middleware chain as an argument. That function returns *another* function that receives the `action` and then triggers any side effects before returning the result of `next(action)`. Side effects can include triggering AJAX requests, logging to the console, and more. Side effects can also happen after `next(action)` is called, like so:

```
const middleware = store => next => action => {
  const result = next(action);
  // side effect using `result`
  return result;
};
```

Creating your own `logger` middleware

You can hand-roll the `logger` middleware you imported above. It should print out the state before and after each dispatch, allowing you to check if your reducers are working as expected. This middleware should:

- receive the store as its only argument,
- return a function that receives the `next` middleware,
- which should itself return a function receiving the `action`.

The body of the innermost function is where you want to do your logging. That function should:

- `console.log` the `action`
- `console.log` the result of `store.getState()` (pre-dispatch)

- call `next(action)` to pass the action on to the rest of the middlewares, and eventually, the reducer
- save the `result` of the `next(action)` variable, to be returned later
- `console.log` the new `store.getState()`
- return the saved `result`

```
const logger = store => next => action => {
  console.log('Action received:', action);
  console.log('State pre-dispatch:', store.getState());

  let result = next(action);

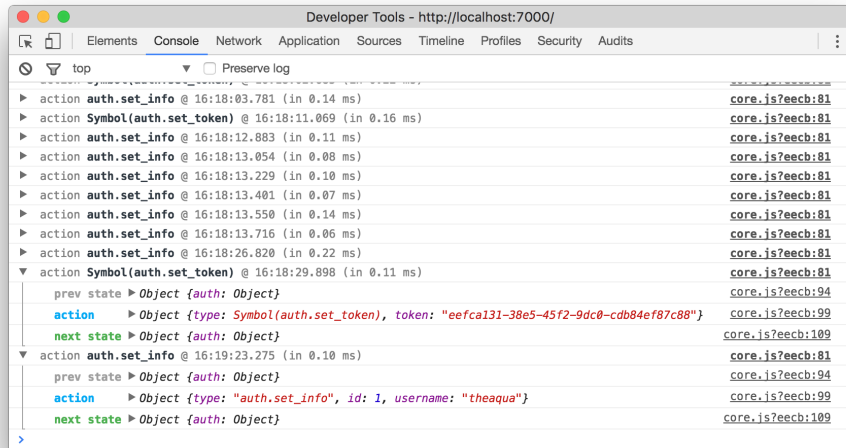
  console.log('State post-dispatch:', store.getState());

  return result;
};
```

Now, whenever you dispatch an action, you'll see its effect on the store.

Installing and applying the `redux-logger` middleware

As you move forward with Redux, you'll want to have access to your store's state for debugging purposes. Including the `redux-logger` npm package and adding it as a middleware gives you access (through the console) to the previous state, action, and next state with each dispatch. This is incredibly convenient for debugging purposes and avoids such unpleasantness as attaching the `store` to the `window`.



Follow the example below to include it in your projects:

- Include the `redux-logger` package:

```
npm install redux-logger
```

- Pass an instance of `redux-logger` to `applyMiddleware` when creating your store:

Note: `logger` must be the last middleware passed into `applyMiddleware`, otherwise it will log the thunk and any involved promises. You'll learn about thunks and `react-thunk` in the next article.

```
// ./src/store.js

import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import logger from 'redux-logger';

import rootReducer from './reducers/rootReducer';

const configureStore = (preloadedState = {}) => {
  return createStore(
    rootReducer,
    preloadedState,
    applyMiddleware(thunk, logger),
  );
};

export default configureStore;
```

What you learned

In this article, you learned how to use the React-Redux library's

`applyMiddleware` function to configure one or more middleware when creating a store.

Thunks

One of the most common problems you need middleware to solve is asynchronicity. When building web applications that interact with a server, you'll need to request resources and then dispatch the response to your store when it eventually gets back.

While it's possible to make these API calls from your components and dispatch synchronously on success, for consistency and reusability it's preferable to

have the source of every change to our application state be an action creator. Thunks are a new kind of action creator that will allow you to do that.

When you finish this article, you should be able to write a thunk action creator to make an asynchronous request to an API and dispatch an action when the response is received.

Looking at how thunks work

Rather than returning a plain object, a thunk action creator returns a function. This function, when called with an argument of `dispatch`, can then dispatch one or more actions, immediately, or later. Here's an example:

```
const thunkActionCreator = () => dispatch => {
  dispatch({
    type: 'RECEIVE_MESSAGE',
    message: 'This will be dispatched immediately.'
  });

  setTimeout(() => dispatch({
    type: 'RECEIVE_MESSAGE',
    message: 'This will be dispatched 1 second later.'
  }, 1000));
}
```

This is great, but without custom middleware it will break as soon as the function action hits your reducer. You need middleware to intercept all actions of type `function` and then trigger the dispatch:

```
// ./src/middleware/thunkMiddleware.js

const thunk = ({ dispatch, getState }) => next => action => {
  if (typeof action === 'function') {
    return action(dispatch, getState);
  }
  return next(action);
};

export default thunk;
```

Notice how the `getState` function is passed into the `action` in case your asynchronous action creators need access to your application state.

Then you'd apply your custom middleware to your store:

```
// ./src/store.js

import { createStore, applyMiddleware } from 'redux';
import logger from 'redux-logger';

import rootReducer from './reducers/rootReducer';
import thunk from './middleware/thunkMiddleware';

const configureStore = (preloadedState = {}) => {
  return createStore(
    rootReducer,
    preloadedState,
    applyMiddleware(thunk, logger),
  );
};

export default configureStore;
```

That's it! Now that you have all the pieces, you're ready to review a more concrete example.

Reviewing a concrete example

Much like the logger from the previous article, thunk middleware is available as the `redux-thunk` library.

The middleware you just wrote is almost the entire original library! ([Check out the source code](#)). For more on thunks and handling asynchronicity in Redux, you can take a look at [this interesting SO post from the creator](#).

Start by using npm to install the `redux-thunk` package:

```
npm install redux-thunk
```

Then apply the middleware to your store:

```
// ./src/store.js

import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import logger from 'redux-logger';

import rootReducer from './reducers/rootReducer';

const configureStore = (preloadedState = {}) => {
  return createStore(
    rootReducer,
    preloadedState,
    applyMiddleware(thunk, logger),
  );
};

export default configureStore;
```

Imagine that you're updating the Fruit Stand application to use a Node/Express API for data persistence. You would use a `fetchFruits` thunk action creator to

retrieve the list of fruits from the API:

```
// ./src/actions/fruitActions.js

import { FRUIT_STAND_API_BASE_URL } from '../config';

export const RECEIVE_FRUITS = 'RECEIVE_FRUITS';

export const fetchFruits = () => (dispatch) => (
  fetch(`${FRUIT_STAND_API_BASE_URL}/fruits`)
    .then((res) => res.json())
    .then((data) => {
      dispatch(receiveFruits(data.fruits));
    })
);

const receiveFruits = (fruits) => {
  return {
    type: RECEIVE_FRUITS,
    fruits,
  };
};
```

Notice that the Fetch API is used to make an HTTP request to the `/fruits` API endpoint. When the promise returned from the `fetch` method call resolves, the `res.json` method is called to parse the JSON into JavaScript objects, which in turn is dispatched to the store using the `receiveFruits` action creator. The `receiveFruits` action creator returns an action of type `RECEIVE_FRUITS` that includes the `fruit` payload.

In the `fruitReducer`, the `RECEIVE_FRUITS` case clause simply returns the `action.fruits` payload as the new state:

```
// ./src/reducers/fruitReducer.js

import { RECEIVE_FRUITS } from '../actions/fruitActions';

const fruitReducer = (state = [], action) => {
  Object.freeze(state);
  switch (action.type) {
    case RECEIVE_FRUITS:
      return action.fruits;
    default:
      return state;
  }
};

export default fruitReducer;
```

To load the fruits from the API when the React application starts up, you can update the `index.js` file to dispatch the `fetchFruits` thunk action creator after creating the store:

```
// ./src/index.js

import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';

import './index.css';
import App from './App';
import configureStore from './store';
import { fetchFruits } from '../actions/fruitActions';

const store = configureStore();
store.dispatch(fetchFruits());

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
);
```

Adding configuration for the API base URL

The `FRUIT_STAND_API_BASE_URL` variable (imported at the top of the `fruitActions.js` file) is defined in the `config.js` file:

```
export const FRUIT_STAND_API_BASE_URL = process.env.REACT_APP_FRUIT_STAND_API_BAS
```

And the `REACT_APP_FRUIT_STAND_API_BASE_URL` environment variable is defined in an `.env` file (located in the root of the React project):

```
REACT_APP_FRUIT_STAND_API_BASE_URL=http://localhost:8080
```

Adding configuration for the API base URL keeps you from having to hard-code a value that'll change between environments.

Reviewing a completed Fruit Stand example

To review and run a completed Fruit Stand example application that utilizes middleware and thunks to support asynchronous interaction with a backend API, clone the [redux-fruit-stand-examples](#) repo.

After cloning the repo, open a terminal and browse to the `fruit-stand-redux-with-react-middleware-thunks` folder.

Running the API

To run the Node/Express API application, complete the following steps:

1. Within the `backend` folder, add an `.env` file based upon the `.env.example` file.
2. Use the following SQL statements to create a PostgreSQL database and user:

```
create database fruit_stand;
create user fruit_stand_app with encrypted password '«a strong password for the f
grant all privileges on database fruit_stand to fruit_stand_app;
```

3. From a terminal, browse to the `backend` folder and run the following commands to apply the Sequelize migrations and seed data:

```
npx dotenv sequelize db:migrate
npx dotenv sequelize db:seed:all
```

4. Start the application using `npm start`.

Running the React application

From the `frontend` folder, run the command `npm install` to install the project's dependencies. Then use the command `npm start` to run the Fruit Stand application.

This Fruit Stand example application is a React application created by the Create React App tooling. When running the application using `npm start`, the application should automatically open in your default browser. If it doesn't, you can manually browse to `http://localhost:3000/` to view the application.

What you learned

In this article, you learned how to write a thunk action creator to make an asynchronous request to an API and dispatch an action when the response is received.

Advanced Containers

While you learned in an earlier article that you should aim to have very few containers, there are exceptions. When you finish this article, you should be able to describe a situation where defining multiple containers for a single component is advantageous.

Knowing when to break the rules

Separating your concerns with presentational and container components allows you to reuse presentational components where it makes sense, rather than duplicating code. If a presentational component needs different data in each situation, though, you may need more containers. By creating more container components, you

can render the same presentational component with each of those containers to suit different needs.

Consider a form component that may either *create* or *edit* a post. The form itself looks and works the same in both cases; it has a few inputs and a submit button. The use cases differ, though, in that the edit form needs to map state from the store to its props, while the create form does not. Furthermore, the edit form will need to dispatch a different action when the form submits than the create form will, as well as request the object from our backend.

As you go through the code snippets below, read the comments carefully.

Here's the presentational component, `PostForm` :

```
// PostForm.js

import React from 'react';

class PostForm extends React.Component {
  constructor(props) {
    super(props);
    // set up initial state
    this.state = this.props.post; // a Post object has a title and a body
  }

  static getDerivedStateFromProps(props, state) {
    // if we get a different post in props, we'll need to set state
    if (props.post.id !== state.id) {
      return props.post;
    }
  }

  update = (field) => {
    return (e) => {
      this.setState({ [field]: e.target.value });
    };
  }

  handleSubmit = (e) => {
    e.preventDefault();
    // `submit` will be a thunk action that presumably creates or edits a post
    this.props.submit(this.state);
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Title
          <input
            type="text"
            onChange={this.update("title")}
            value={this.state.title}
          />
        </label>
      </form>
    );
  }
}
```

```

    <label>
      Body
      <input
        type="text"
        onChange={this.update("body")}
        value={this.state.body}
      />
    </label>

    <button>Submit Post</button>
  </form>
);
}
}

export default PostForm;

```

You can see that `PostForm` is expecting two things in props: a `post` object and a `submit` function. The container will have to define these, since right now, this form can't actually do anything. Give it the ability to create a post:

```

// CreatePostFormContainer.js

import { connect } from 'react-redux';
import PostForm from './PostForm';
import { createPost } from '../actions/postActions';

const mapStateToProps = state => {
  return {
    post: { title: '', body: '' } // a default blank object
  };
};

const mapDispatchToProps = dispatch => {
  return {
    submit: post => dispatch(createPost(post))
  };
};

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(PostForm);

```

So far, this is nothing new. Now, wherever you need a form to create a post, you can render `CreatePostFormContainer` by importing from the above file.

But what about editing? This is a little trickier, because you need more information from the store - so you'll need a [higher-order component](#) to help you out. Higher-order components are a useful React pattern that essentially uses a component to render another component, usually to handle some sort of work and pass in data. This pattern allows us to keep your components small and modular. Here, you'll use a higher-order component to fetch the post you want to edit and pass it into the `PostForm`:


```
// EditPostFormContainer.js

import React from 'react';
import { connect } from 'react-redux';
import PostForm from './PostForm';
import { fetchPost, updatePost } from '../actions/postActions';
import { selectPost } from '../reducers/postSelectors';

const mapStateToProps = (state, ownProps) => {
  const defaultPost = { title: '', body: '' };
  const post = selectPost(ownProps.match.params.postId) || defaultPost;
  // get the post this route is asking for
  // (assuming here that this component is being rendered by a route)
  // if you don't have the post in state yet, return a blank post so PostForm does
  return { post };
};

const mapDispatchToProps = dispatch => {
  // an edit form will need to fetch the relevant post, but the PostForm shouldn't
  // you'll handle this problem with a higher-order component, EditPostFormContainer
  return {
    fetchPost: id => dispatch(fetchPost(id)),
    submit: post => dispatch(updatePost(post))
  };
};

class EditPostForm extends React.Component {
  // this is the higher-order component made to handle the fetch

  componentDidMount() {
    // do the fetching here so that PostForm doesn't have to
    this.props.fetchPost(this.props.match.params.postId);
  }

  render() {
    // destructure the props so you can easily pass them down to PostForm
    const { post, submit } = this.props;
    return <PostForm post={post} submit={submit} />;
  }
}

// now `connect` it to the Redux store
```

```
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(EditPostForm);
```

The result here is that we can render a `CreatePostFormContainer` wherever you want a form to create a post, and an `EditPostFormContainer` wherever you want to edit a post. Both components will render a `PostForm`, but each will have different functions. The `PostForm` also gets to be very simple and make almost no decisions. This helps keep your code DRY and modular.

You can use this pattern with any presentational component that needs to be connected to the store, but may need entirely different data to perform different functions.

What you learned

In this article, you learned about a situation where defining multiple containers for a single component is advantageous.

Redux Developer Tools

Redux has its own special set of developer tools. They allow you to do things like inspect your application state in real time as you use your app, or cancel an action to see a live recalculation of the state as if that action had never been dispatched. They require only a few minutes of setup, and can be well worth the effort.

Instructions

1. Install the [chrome extension](#).
2. Install the npm package into your project:

```
npm install redux-devtools-extension
```

3. Make the following changes to your `./src/store.js` file.

If you're *not* using middleware:

```
// ./src/store.js

import { createStore } from 'redux';
+ import { devToolsEnhancer } from 'redux-devtools-extension';

import rootReducer from './reducers/rootReducer';

const configureStore = () => {
  return createStore(
    rootReducer,
+   devToolsEnhancer()
  );
};

export default configureStore;
```

Or if you're using middleware:

```
// ./src/store.js

import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import logger from 'redux-logger';
+ import { composeWithDevTools } from 'redux-devtools-extension';

import rootReducer from './reducers/rootReducer';

const configureStore = () => {
  return createStore(
    rootReducer,
+   composeWithDevTools(applyMiddleware(thunk, logger))
-   applyMiddleware(thunk, logger)
  );
};

export default configureStore;
```

Use

Now that you've set up the Redux dev tools, you can try them out. You'll use one of the Fruit Stand application examples. If you haven't already, clone the [redux-fruit-stand-examples](#) repo.

After cloning the repo, open a terminal and browse to the `fruit-stand-redux-with-react-official-bindings` folder. Run the command `npm install` to install the project's dependencies. Then use the command `npm start` to run the Fruit Stand application.

This Fruit Stand example application is a React application created by the Create React App tooling. When running the application using `npm start`, the application should automatically open in your default browser. If it doesn't, you can manually browse to `http://localhost:3000/` to view the application.

Open the project into your code editor and complete the above set up steps.

You should see an atom (a nucleus with electrons) icon on your Chrome toolbar, and if you've set up the Redux dev tools correctly it should now be green. Click on it. When the Redux dev tools open, click one of the buttons on the very bottom left to open them in a new window.

Now try adding some fruit. This will cause actions to be dispatched. You should see those actions popping up in the Redux dev tools. You can click on them to cancel them and you should see the state recalculated in real time.

The Redux dev tools have some other handy features, so click around and explore!

Resources

- [Redux Dev Tools - Chrome Extension](#)
- [Redux Dev Tools - Github Page](#)
- [Redux Dev Tools - Demo](#)

Redux-Based Pokedex Project

Instead of building a new application, you will spend time refactoring an existing application, one that is not yours. Navigating around someone else's code is an interesting way to learn what to do and what *not* to do. The starter application is in a little bit of a mess and needs your help to get unmessed.

By the end of this walk-through, you will be able to:

- Describe the Redux data cycle
- Explain a *reducer*
- Configure a React application to use Redux

- Use connected components to access Redux state
- Use composed reducers to simplify state management
- Configure a React application to use the Redux development tools

Getting started

You'll need the backend of the application.

If you have *not* cloned it, please clone the repository from <https://github.com/appacademy-starters/pokedex-backend>.

If you have cloned it, please get the latest (things sometimes change) by going to the repository in your Terminal and typing `git pull origin master`. This will get the latest code for you.

You'll need the starter application. Please clone the repository from <https://github.com/appacademy-starters/redux-pokedex-starter>.

Tour the application

This application is a Heroku-deployable React application. Here are the files that are in it. Hopefully, you find nothing surprising about the file layout or the intent of each file. Take a moment to look in each file to get the lay of the land. (Really, take a look in each file. You may feel like jumping right into it, but looking at other people's code is helpful. This is one of the benefits to the pair-programming learning style at App Academy.)

├─ package-lock.json	- The NPM lock file
├─ package.json	- The NPM file
├─ public	
│ └─ index.html	- The page that gets served to the browser
├─ server.js	- A very light-weight server for Heroku
└─ src	
├─ App.js	- The main application component
├─ Fab.js	- A floating action button component
├─ LoginPanel.js	- The form that shows the login
├─ LogoutButton.js	- A component for logging out
├─ PokemonBrowser.js	- The component that shows the list and detail
├─ PokemonDetail.js	- The component that shows the detail of a Pokemon
├─ PokemonForm.js	- The form to create a new Pokemon
├─ config.js	- Configuration variables
├─ index.css	- Styling for the application
└─ index.js	- The main entry point for Webpack

Start the backend with `npm run dev`. Start the React application with `npm run dev`. Make sure it runs. It looks for a local backend at `http://localhost:8000`, so make sure that's where the backend is running.

Install Redux and DevTools

If you haven't already, install [Redux DevTools](#). During development, you can watch the Redux store handle actions and change state in the timeline.

To use Redux in this application, you need to install it and the connector between Redux and React. You will also want to use asynchronous actions with the Redux store, so you'll want a middleware, one like [Redux Thunk](#).

```
npm install react-redux redux redux-thunk
```

There are more than one asynchronous action-handling middleware out there in the world. Redux Thunk happens to be one of the oldest and widely used.

Whenever you consider installing a library or framework, you should make sure that your existing application meets the expectations. For example, as of the time of this writing, to [install "react-redux"](#), your application needs to support **React 16.8.3** or later. Take a look in the **package.json** file to make sure that an acceptable version of React is listed in there. If not, you will need to upgrade the version of React used by this project by running something like `npm upgrade`.

Setting up the store

The *store* is the object (and supporting objects and functions) that will contain the state of the application. This centralizes the state so that, presumably, you can better reason about it.

There are a couple of ways to organize your state management code, each with their own benefits. Redux has a list of different articles about [this very topic](#). You should choose to organize your code in a way that makes sense to your team (or follow any conventions that already exist). This walk-through will follow the [Ducks](#) approach of layout.

In the **src** directory, create a new directory named **store**. In that new directory, create a new file named **configureStore.js**. In that file, put this code. This is boilerplate code and will appear in nearly every application that you have that uses Redux. A description of the contents follows the code block.

```
import { createStore, applyMiddleware, combineReducers, compose } from 'redux';
import thunk from 'redux-thunk';

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

const reducer = combineReducers({

});

const configureStore = initialState => {
  return createStore(
    reducer,
    initialState,
    composeEnhancers(applyMiddleware(thunk)),
  );
};

export default configureStore;
```

(If you're looking in your console after putting this code in there, you'll see an error about not having any valid reducers. That's true, there are no valid reducers. You'll fix that in just a moment.)

On those first two lines, the code imports the stuff it will need to create and configure a store.

- `createStore` is the function from Redux that creates the state-management object
- `applyMiddleware` allows you to plug in extra functionality into the state-management workflow (in this case, Redux Thunk)
- `combineReducers` takes many reducers and combines them into a single one
- `compose` is a function that *composes* functions from right-to-left, that is, it puts them together, the return value of the right-most getting passed to the second right-most, return values getting passed from that to the third right-most, and so on, until the first function in the list returns its final value to the store ([documentation](#) for compose)

- `thunk` is the middleware that will allow you to make asynchronous calls because you can't do that in Redux actions

The fourth line creates a new `composeEnhancers` variable that will be either the Redux DevTools special compose function, or the one from Redux, if DevTools is not installed. This allows browsers that have the DevTools installed to take advantage of watching the changes in the store.

The sixth and seventh lines is just the thing that combines the reducers for the store into a single reducer. More about that later.

The ninth line declares a function that gets exported at the end of the file that creates and configures the store with the `reducer`, the `initialState` passed into the function, and composes the React Thunk middleware and the Redux DevTools, if they exist.

To use this functionality, open **src/index.js** and add two imports to the top of the file.

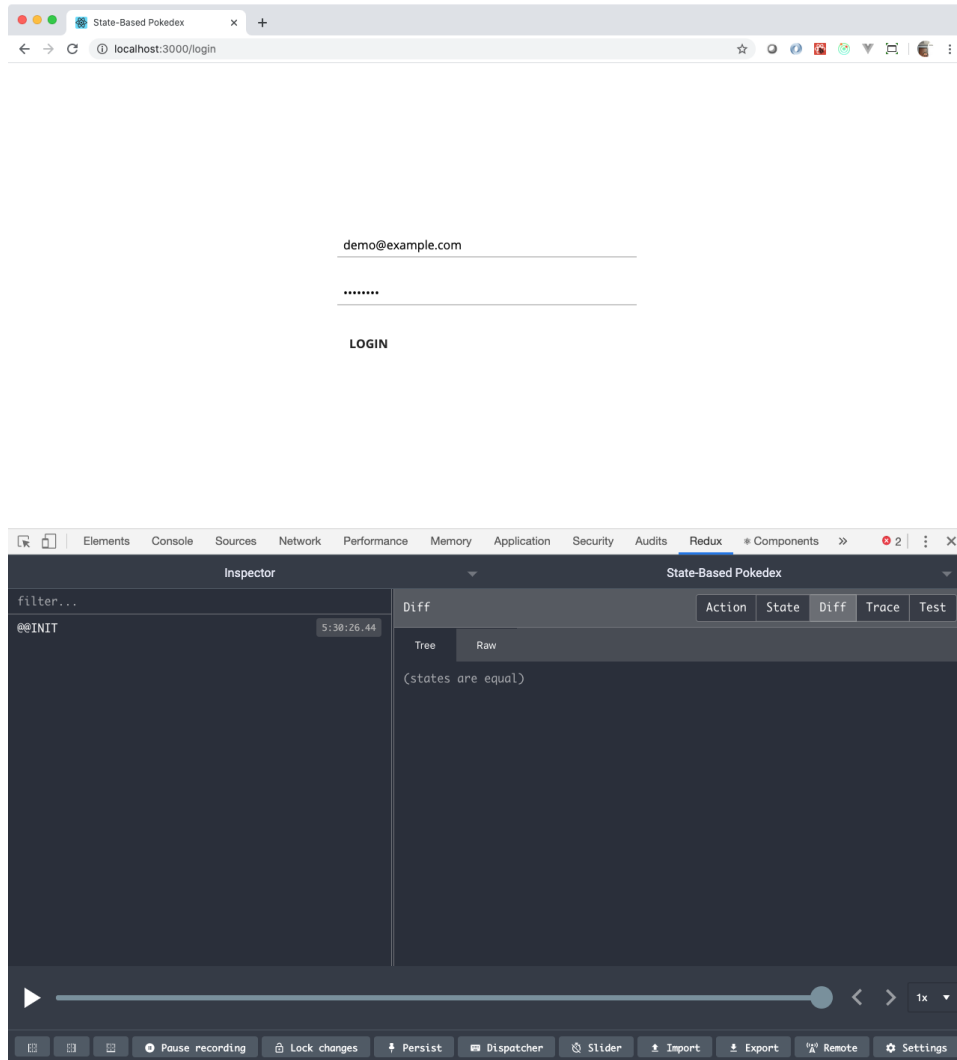
```
import { Provider } from 'react-redux';
import configureStore from './store/configureStore';
```

Those two lines import a Context Provider from React Redux and the function you just created above. Now, under those lines and before the `ReactDOM.render` statement, create a store.

```
const store = configureStore();
```

Finally, wrap the `App` component in a `Provider` component that has a "store" property assigned the value of `store` that you created in the previous code snippet, just as you saw in the *Passing the Store* section of the *Usage with React* article you read in the homework preparing you for these topics.

If everything works, you should be able to view the Redux DevTools in your browser's DevTools environment as a new tab. You should see the "@@INIT" action that was run and an empty state.



Your first action workflow

Now, you will refactor the `LoginPanel` component to put the token it receives from the AJAX call into the Redux store. The rest of the application will remain the same, for now.

Open `LoginPanel` and review it. There are two "interesting" portions of this code, the `render` function which will redirect to the path "/" when a token exists (and show the login form if it does not), and the `handleSubmit` method that makes the actual AJAX call to get the token. These are the things that you will modify to make this work. It's going to seem like a lot of code replacing just a few lines, but this will make your application easier to understand because all of the state changes will be encapsulated in their own area under the **store** directory. You won't have to search through a bunch of components to figure out why the application as a whole is not working.

The first step is to make `LoginPanel` a *connected component*, that is, connect it to the Redux store's pipeline. To do this, follow these steps:

1. Import `connect` from `react-redux` in the import section of the `LoginPanel`.
2. Remove the `export` line at the bottom and replace it with this.

```
const mapStateToProps = state => {
  return {
  };
};

const mapDispatchToProps = dispatch => {
  return {
  };
};

// Yes, this looks funny, but you will often
// see this kind of indentation in others'
// code when using React and Redux.
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(
  LoginPanel
);
```

The last line of the code *connects* the `LoginPanel` to the Redux store. The two functions that you created above it, `mapStateToProps` and `mapDispatchToProps` are functions that you will write to help in translating state and actions for use in your component. The `mapStateToProps` function maps the state of the Redux store to the values that you want to show up in the `props` of the connected component. The `mapDispatchToProps` function maps complicated action calls to simple ones that your component can use.

The first thing you'll need is one of those actions. This is going to be something to let your authentication run to get that token. In the **src/store** directory, create a new file named **authentication.js** which will contain all of the Redux-related stuff to handle authentication:

- the actions,
- the action types,
- the thunks (which are just actions that are allowed to perform calls that have use resources *outside* the function, like AJAX calls or putting

something into local storage), and,

- the reducers.

The steps that you want the application to go through are:

- Make an AJAX call to sign in (that's a *thunk*)
- When the token returns, create an *action* that will send the token to the store
- Create a reducer to put the token in the store

Most of these steps will need something more, so if you see something not defined in one of these steps, you will get to it in a later step.

The thunk

In case you lost track, this goes in **src/store/authentication.js**.

First up, *thunks* are functions that return another function that takes a single function as its argument. The argument is the `dispatch` method used to dispatch actions to Redux. So, for example, the `login` thunk could have this form.

```
export const login = (email, password) => async dispatch => {
  // Dispatch an action, here
}
```

That's pretty weird, if you've never seen that syntax before, those double `=>` signs in there. It's a shortcut to write a function that returns a function. You could also write it like this:

```
export const login = (email, password) => {
  return async dispatch => {
    // Dispatch an action, here
  }
}
```

Once you get used to the double (or triple) `=>` signs, it becomes second nature to write functions that return functions, that way.

Inside the login method, make a `fetch` call to the API using the same `fetch` call found in `LoginPanel`'s `handleSubmit` method. You'll need to import `baseUrl` from the **`src/config.js`** module. You don't have access to the state of the object, just the `email` and `password` that they're passing in through your method call, so change the "body" parameter of the `fetch` from `JSON.stringify(this.state)`, to `JSON.stringify({ email, password })`, .

For now, if the response from the fetch is ok, just get the token out of the response object and log it to the console.

Now, to hook up this thunk to your component, open **`src/LoginPanel.js`** and import the `login` thunk that you just created. At the bottom in the `mapDispatchToProps` function, add a "login" key that is a function that takes an email and password, and then dispatches the `login` thunk with those values.

```
return {
  login: (email, password) => dispatch(login(email, password))
};
```

Now, there's a "login" property on the `props` handed to the `LoginPanel` component. The value of the "login" property is a function that takes an *email* and a *password*. Those then get handed to the `login` thunk which returns a function to the `dispatch` function of Redux. It's functions all the way down.

Now, delete everything from the `handleSubmit` method *except* the line that prevents the default action. Replace it with a call to the function in the "login" property of the `props` like this.

```
this.props.login(this.state.email, this.state.password);
```

After the application refreshes, you should be able to click the *Login* button on the screen and see token appear after the AJAX call completes.

Now that you have the token, you need to dispatch another action, one that isn't created yet, to set the token in the state so it can be used elsewhere. Remove the `console.log` statement in your `login` thunk and replace it with an invocation of the `dispatch` method that your thunk gets, dispatching an action creator named `setToken` with the token as its argument.

```
dispatch(setToken(token))
```

Now, it's time to create that action.

The action

Actions are just plain objects that have, at a minimum, a "type" property. Action creators are just plain functions that return actions.

Below your import section and above the `login` thunk of your code in **`src/store/authentication.js`**, create a constant named `SET_TOKEN` and set it equal to the string `'pokedex/authentication/SET_TOKEN'`. You could make this string *anything*, it just needs to be unique within your application. This is merely the [Ducks](#) convention.

Now, create a function `setToken` that takes a token as its one parameter, and returns an object that has a "type" property set to the `SET_TOKEN` constant and a "token" property set to the value passed into the parameter. Export the `setToken` function.

If you've done everything correctly to this point, when you click the *Login* button, you should now see an action appear in the Redux DevTools with the string that you set the `SET_TOKEN` constant to.

 Redux DevTools with SET_TOKEN action

Now, you need to tell Redux how to handle that action with a reducer.

The reducer

Somewhere in the **src/store/authentication.js** file, export a `reducer` function as the default value of the module. The `state` parameter gets a default value of an empty object because Redux does not like `undefined` values returned from reducers. Redux *will* call your reducer when it creates the store with `undefined` just to mess with you.

```
export default function reducer(state = {}, action) {  
  // Your code in here.  
}
```

In that reducer, you want to check if the "type" property of `action` is equal to the `SET_TOKEN` constant. Every reducer is called with every action, so you have to do this check. If it is, then you should return an object with the token in it. If it doesn't, then just return the state unaltered. Idiomatic Redux usually uses a `switch` statement to do this with each `case` statement handling a different action type.

```
export default function reducer(state = {}, action) {  
  switch (action.type) {  
    case SET_TOKEN: {  
      return {  
        ...state,  
        token: action.token,  
      };  
    }  
  
    default: return state;  
  }  
}
```

Adding the reducer to the store

Now that you have the reducer handling the action, you must add it to the reducer in **src/store/configureStore.js**. Open up that file and import the default value as `authentication` from the **src/store/authentication.js** module.

```
import authentication from './authentication';
```

Then, in the `combineReducers` invocation, add `authentication` as a key and value.

```
const reducer = combineReducers({  
  authentication  
});
```

After the page reloads, when you click the *Login* button, you should now see the authentication and token appear in the right pane of the Redux DevTools. (You may need to select the action in the left pane.)

Using the token

Now, you've come full circle. You're going to use the token in the store to inform the `LoginPanel` that it needs to close. In the `render` function, the code uses `this.state.token` to determine whether or not to redirect. You want that to use `this.props.token`, now, so change it to that.

To get the token value from the state of the Redux store into the `props`, you use `mapStateToProps` function that you created below. Change it to read like this, now, which takes the value of the token stored in the state and puts it into the "token" property of what will be passed to `LoginPanel` in its `props`.

```
const mapStateToProps = state => {
  return {
    token: state.authentication.token,
  };
};
```

Now, when you click the *Login* button, it redirects the application back to `/`. However, because `App` relies on its own state to get updated from a call to its `updateToken` method, it goes into an recursive loop of redirecting back and forth between the `App` component and the `LoginPanel`. You'll fix that in the next section.

More State Moving

The application is cleaner, but now has that bug of infinite redirecting. You will first fix that.

Fixing the loop

In the `App` component, it would change state in response to the `LoginPanel` calling the `updateToken` method. The `LoginPanel` no longer does that because it dispatches its information to a thunk that makes the AJAX call and, in turns, dispatches an action that updates the Redux store. Then, `LoginPanel` uses the value from the store to no that something good has happened. You need to have `App` do the same.

Since `updateToken` is no longer used, remove the method and all of its uses from the `App` component.

When you have that done, you may notice that the route that shows the `LoginPanel` looks like this.

```
<Route path="/login"
  render={props => <LoginPanel {...props} />} />
```

That is rendering `LoginPanel` and passing its props to it. That's just like using the "component" property of the `Route` component, so change it to use that instead of the more expensive "render" property.

```
<Route path="/login" component={LoginPanel} />
```

You're still in a render loop, so that hasn't fixed it. Time to connect `App` to the Redux store.

1. Like you did in `LoginPanel`, import the `connect` function from the "react-redux" module.
2. Then, just like you did in `LoginPanel`, declare `mapStateToProps` and `mapDispatchToProps` functions after the component.
 - In the `mapStateToProps` function, map the token to a property named "token" identically to the way it is done in `LoginPanel`.
 - You have no actions, right now, so leave `mapDispatchToProps` just returning an empty object.
3. Finally, connect `App` to the Redux store using those functions.

That puts the value of the token into the props. In the actual `App` component, now find everywhere that uses `this.state.token` and replace it with `this.props.token`.

Try logging in, again. Still doesn't work. That's because `App` relies on another setting in its state named "needLogin" which is just a Boolean value that is basically the opposite of whether the token has a value. If there is a

token, there is no need to login. If there is no token, there is a need to login. You can figure that out in the `mapStateToProps` function! Create another property, one named "needLogin", in the object returned from the function. If there *is* a value in `state.authentication.token`, then set it to `false`. If there *is no* value (or an empty string) in `state.authentication.token`, then set "needLogin" to `true`. Once you have that, find every use of `this.state.needLogin` and replace it with `this.props.needLogin`.

You have now fixed the problem with the infinite redirects! And, once you log into the application, you can check that the `App` component *is* rendering the `PokemonBrowser` component. However, it is not showing anything. That's because the `App` component calls its `loadPokemon` method in the `componentDidMount` method. There's no token at that time in its state because the token now comes from Redux.

What would be great is if, after logging in, the `login` action loaded the Pokemon for you, put them in the store, and `PokemonBrowser` could just use them directly.

Oh, that's what you'll do next.

Getting the list of Pokemon

Now, you need some actions, thunks, and reducers for Pokemon, not authentication. Create a new file **src/store/pokemon.js**. In there, create the following items.

- An action type named `LOAD` with a value of 'pokedex/pokemon/LOAD'
- An action creator named `load` that takes in a list of Pokemon and creates an action with the type of `LOAD` and the list of Pokemon in it
- A reducer that checks for the "LOAD" action and adds the list of Pokemon to the state (and returns just the state if the action is not "LOAD")
- Make sure the reducer has a default value for its `state` parameter

Now, you need to make a thunk that will make the AJAX call. Call your think "getPokemon". The thunk needs the token from the state to make its API call. The function that gets the "dispatch" parameter can also get a second parameter, a function conventionally called "getState". Your thunk could look like this.

```
export const getPokemon = () => async (dispatch, getState) => {
  const { authentication: { token } } = getState();
  // AJAX call
  // Handle response
};
```

Then, in **configureStore.js**, import the default reducer and add it to the `combineReducers` argument as the "pokemon" property next to "authentication".

```
const reducer = combineReducers({
  authentication,
  pokemon,
});
```

Now, you just need to kick off that AJAX call. It seems only reasonable that the component that actually needs it kicks it off, the `PokemonBrowser` component. In `PokemonBrowser`, do the following:

- Import the `connect` function from "react-redux"
- Import the `getPokemon` thunk you just created
- Create a `mapStateToProps` function that maps the list of Pokemon from `state.pokemon.list` (or whatever you called your property in the reducer) to a property named "pokemon"
- Create a `mapDispatchToProps` function that returns an object with a property named "getPokemon" that dispatches the `getPokemon` thunk you imported
- Connect your component to the Redux store using `connect`, `mapStateToProps`, and `mapDispatchToProps`
- Add a `componentDidMount` method to the `PokemonBrowser` component and call `this.props.getPokemon()` from it

Now, when you log into the application, you should see two actions in your Redux store!

 Pokemon LOAD action in Redux DevTools

More importantly, you should see the list of Pokemon in the `PokemonBrowser` actually appear in the browser.

Clean up the App component

Previously, all of the fetching logic for the list of Pokemon was handled by the `App` component. You can now get rid of the `loadPokemon` method and clean up any calls to it. Also, anywhere that refers to `this.state.pokemon` or using `setState` to update the "pokemon" property, you should get rid of all of that. This makes the `handleCreated` method empty, so get rid of that and all references to it, too, in the `App` component.

Because the `App` component doesn't make any AJAX calls, anymore, it doesn't need the token from the state. Remove the "token" property in `mapStateToProps` and everywhere `token` is used in the `App` component. Include the call to local storage, too, in the deleting of things. You can remove the import of `baseUri`, too, because there are no AJAX calls in the file.

Now, because `cProps` in the `render` method is empty, delete it and its uses in the `PrivateRoute` components below it. Now that the code is not using `cProps`, you can delete the parameter and its use from the `PrivateRoute` component on line 8 (or so) of the **src/App.js** file.

Storing the token in local storage

Actions that use outside resources like AJAX calls and local storage *must* be created in thunks. Back in the **src/store/authentication.js** module, do two things:

- Create a constant named `TOKEN_KEY` and set it equal to some non-empty string
- In the `login` thunk, between getting the token from the response and dispatching the `setToken` action, write the token to local storage using the constant `TOKEN_KEY`

Reading the token out of local storage

You need a new thunk to do this. Create a thunk named `loadToken` that takes no parameters, and returns a function that accepts a "dispatch" parameter. Then, implement it to read the value from local storage using the `TOKEN_KEY` constant. If a value comes back, have it dispatch the `setToken` action.

```
export const loadToken = () => async dispatch => {  
  // Read the token from local  
};
```

In the `App` component, import the `loadToken` thunk. Use it in the `mapDispatchToProps` by mapping the dispatch of the `loadToken` thunk to a property of the same name.

```
const mapDispatchToProps = dispatch => {  
  return {  
    loadToken: () => dispatch(loadToken()),  
  };  
};
```

Invoke that `loadToken` method in the `componentDidMount` method of the `App` component.

```
async componentDidMount() {  
  this.setState({ loaded: true });  
  this.props.loadToken();  
}
```

The moment you do that, the page should refresh and you should see the Pokemon browser rather than the login form.

You're halfway home!

Making Decisions About State

Now, it's time to log out of the application. Do that with the following steps.

The steps that the application will take are these:

1. Someone clicks the logout button
2. The `LogoutButton` component dispatches a thunk
3. The thunk makes the AJAX call to logout
4. If that AJAX call succeeds, remove the token from local storage and dispatch an action to remove the token from the store
5. Redux will invoke a reducer that removes the token from the store
6. The `LogoutButton` will redirect the application back to `"/login"`

In `src/store/authentication.js`:

- Create a new action type named `REMOVE_TOKEN`
- Create a new action creator that returns an action with just the "type" property set to the value of `REMOVE_TOKEN`
- Create a thunk named `logout` that
 - makes an AJAX call to the API to `DELETE /api/session` (using the token in the state)
 - if the response is ok, then
 - removes the item from local storage with the key `TOKEN_KEY`
 - dispatches the `removeToken` action

- Handles the `REMOVE_TOKEN` action type in the reducer by creating a new object that does *not* have the "token" key in it and returning that.

```
case REMOVE_TOKEN: {
  const newState = { ...state };
  delete newState.token;
  return newState;
}
```

Remember that handlers in reducers *must* return new objects if they want to modify the state.

In `src/LogoutButton.js`:

(If you make a mistake with this and get into an inconsistent state, just delete all of the contents of your local storage and refresh your browser.)

- Import `connect` from "react-redux"
- Import the `logout` thunk you just created
- Create the `mapStateToProps` and set a property named `loggedOut` to `true` if the token in the state is empty, and `false` if there is a value for the token in the state
- Create the `mapDispatchToProps` and set the "logout" property equal to a function that dispatches the result of the `logout` thunk you imported
- In the `LogoutButton`'s method named "logout", instead of making an AJAX call, have it call `this.props.logout()`, instead
- Get rid of the
- Get rid of initializing the state in the constructor
- Change the use of `this.state.loggedOut` to `this.props.loggedOut`
- Remove any unused imports

If you check the console, now, you'll see that Babel is reporting a "useless" constructor. Sure enough, it is. `LogoutButton` no longer has any state, so there's no reason to leave it as a class-based component. Convert it to a

function-based component. If you've followed these instructions, your `LogoutButton` should end up looking something like this.

```
const LogoutButton = props =>
  props.loggedInOut ?
    <Redirect to="/login" /> :
    <div id="logout-button-holder">
      <button onClick={props.logout}>Logout</button>
    </div>
  ;
```

The Rest Of It

You have now been given instructions on how to refactor components from managing global application state to putting it in Redux. There are two more pieces left, the "select the current Pokemon" functionality and the "create a new Pokemon" functionality. Refactor the application so those are Redux-supported, as well.

Select the current Pokemon

The place to start, here, is to determine how the click of the navigation item on the left gets handled. It's a `NavLink`, so the `BrowserRouter` in the `App` component handles that by routing to the `PokemonBrowser` with the route parameters. The `PokemonBrowser` then routes to the `PokemonDetail` with a `Route` component. In the `PokemonDetail` component, if the value of the `this.props.match.params.id` changes, then the `loadPokemon` method is called which, in turn, makes an AJAX call. And, there it is! The AJAX call.

This is like everything else, create a thunk, an action type, yada yada yada.

- Create a thunk (similar to what you did for logging in) to load the current Pokemon that
 - accepts an id
 - loads the Pokemon from an AJAX call
 - dispatches a "set current Pokemon" action
- Create a reducer that handles the "set current Pokemon" action by adding it to the state
- Connect the `PokemonDetail` to the Redux store by
 - mapping the current Pokemon information in the state to its props, and
 - mapping the "load the current Pokemon" thunk to its props with a `dispatch` call (don't forget the id parameter)

Creating a new Pokemon

This is very similar to the login stuff you did with `LoginPanel`. In the `PokemonForm`, have

- the `componentDidMount` method call a thunk to load the Pokemon types
- the `handleSubmit` method call a thunk to post the form information to the API

In moving the Pokemon type fetching from the state to the props, you may end up getting an error that there is no method "map" of undefined. If that's the case, in the reducer in your **src/store/pokemon.js** file, have the default state include an array for the "types" property.

```
// CODE SNIPPET
export default function reducer(state = { types: [] }, action) {
```

That's the power of default parameters and initial state!

The action types, action creators, and thunk created to do this should go into the **src/store/pokemon.js** module. When the AJAX call succeeds to create the

new Pokemon, have it *then* dispatch the `getPokemon` thunk to get a new list of Pokemon. Redux and React will add a new Pokemon to the end of the list. That's why you have to provide the "key" property in lists of things, so that React will efficiently determine if something in the list needs to get changed, added, or deleted.

The only "new" part, here, is the coordination between `PokemonForm` and `PokemonBrowser` to determine if it should show a form. This is up to you to decide, if showing the create form is part of the global application state (and should exist in the Redux store), or if it is part of the "local" state between the two components and be managed by `PokemonForm` invoking a function passed to it by `PokemonBrowser`. The solution choose the former solution.

Bonus: Extend the functionality

Think about adding

- A *Cancel* button on the form that hides it
- Error messages for the forms when something bad happens

Bonus: Connected React Router

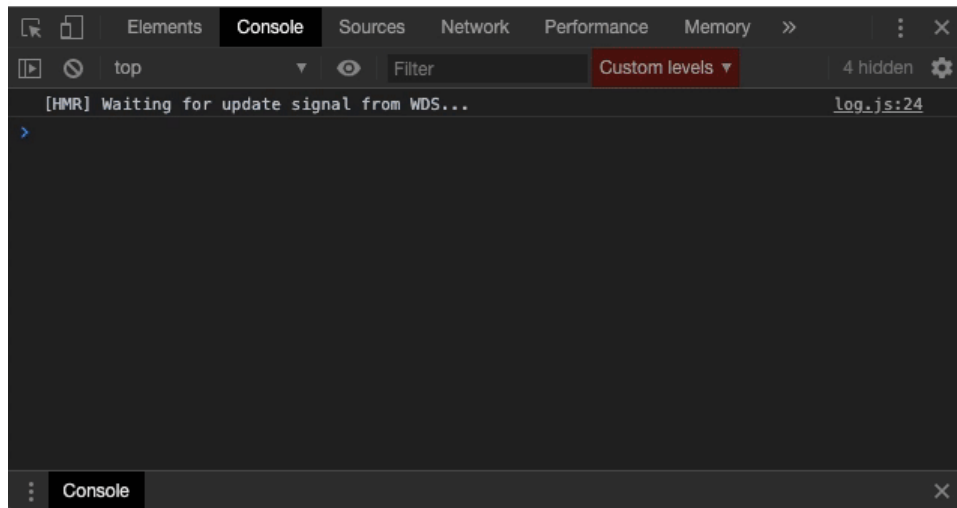
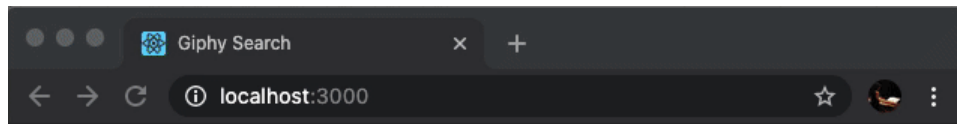
Rather than relying on `Redirect` routes in your application, you can use actions to manage the URL of your application. Install [Connected React Router](#) and remove all `Redirect` components from the application, replacing them with dispatched `push` actions. Check out the [How to navigation with Redux action](#) article in the Connected React Router documentation.

Today's project will help you become more comfortable with the full Redux cycle! You will build out a single Redux cycle for a Giphy search tool.

When a user enters a search query, a fetch action will use the search endpoint from the Giphy API to return the fetch response. The action will then be dispatched to update the application's global state provided by the Redux store. Your application will then use a slice of state to render an index of GIF results.

Your completed project will look something like this:

Giphy Search Project



Phase 0: Set up the project

You'll begin by cloning this repository containing basic skeleton files and the conventional frontend folders (`actions` , `components` , `reducers` , and `util`).

```
git clone https://github.com/appacademy-starters/react-redux-giphy-starter.git
```

Take a moment to familiarize yourself with the file structure. Look inside all the frontend files and note the `TODO` notes in each file skeleton. Throughout the next phases of the project, you'll finish all the tasks listed in the `TODO` notes to create your very own Giphy search app!

After you have reviewed the `TODO` notes, run `npm install` to install your application's packages. Note that you have `redux` , `react-redux` , `redux-thunk` , and `redux-logger` already listed as dependencies in your project's `package.json` file. You'll use Redux `thunk` as middleware to connect (or *dispatch*) your fetch results into the Redux store. You'll use Redux `logger` as middleware to automatically console log your dispatched actions.

Component overview

Now that you're acquainted with the file structure, let's map out an overview of the component hierarchy:

```
Root
  AppContainer
    App
      SearchBar
      Gifs
```

- The `Root` component is responsible for providing the component tree with the `Redux store` . It renders the `AppContainer` .
- The `AppContainer` passes the `gifs` slice of state and the dispatched `fetchGifs` action creator as props to the `App` component. It wraps the `App` component to connect the component to the Redux store.
- The `App` component renders the `GifsList` and the `SearchBar` . It uses its `gifs` prop (passed into `App` through `mapStateToProps`) to create and pass a `gifUrls` array as a prop to the `GifsList` . It also uses its `fetchGifs` prop

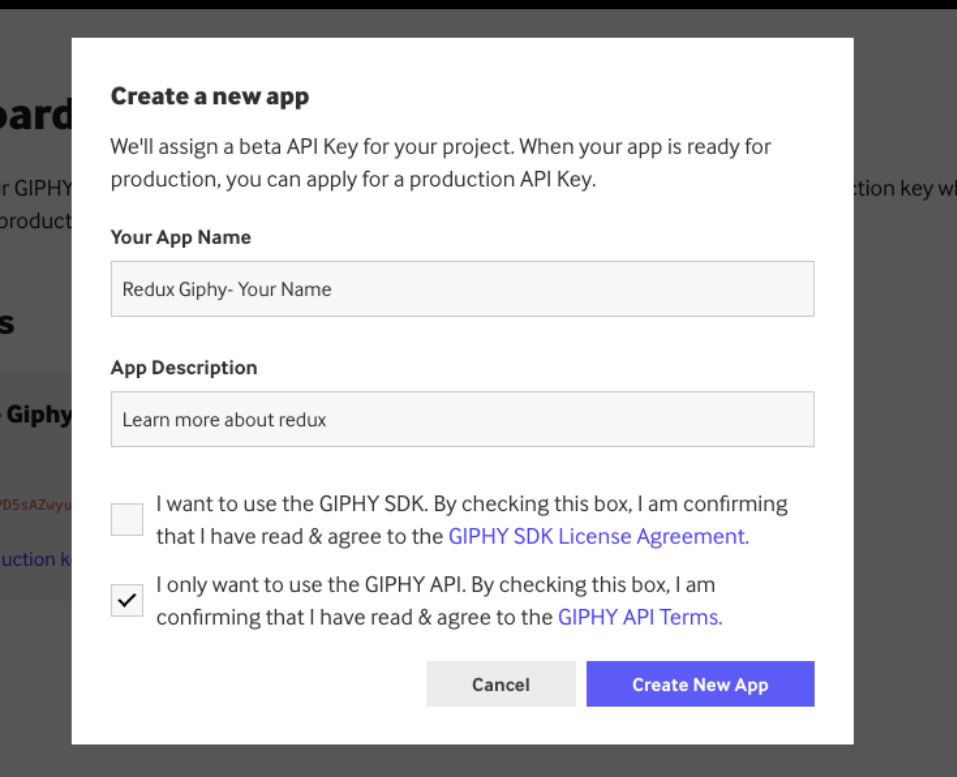
(passed into `App` through `mapDispatchToProps`) as a prop to the `SearchBar` component.

- The `SearchBar` component handles all of the search logic (keeping track of the query and triggering the fetch request on submit).
- The `GifsList` component iterates over its `gifurls` prop, to render an image for each one.

Giphy API key

Before you start, let's create a Giphy API Key to use in your fetch requests to the Giphy API. Get started by [creating a Giphy account](#). Then navigate to the [Giphy API Quick Start Guide](#) and click `Create an App`.

Fill out the form for creating a new app, and only check the option for `I only want to use the GIPHY API`.



The screenshot shows a 'Create a new app' form. At the top, it says 'Create a new app' and explains that a beta API key will be assigned. Below this, there are two text input fields: 'Your App Name' with the placeholder text 'Redux Giphy- Your Name', and 'App Description' with the placeholder text 'Learn more about redux'. There are two radio button options for the SDK type. The first option is 'I want to use the GIPHY SDK. By checking this box, I am confirming that I have read & agree to the GIPHY SDK License Agreement.' and it is currently unchecked. The second option is 'I only want to use the GIPHY API. By checking this box, I am confirming that I have read & agree to the GIPHY API Terms.' and it is checked. At the bottom right, there are two buttons: a grey 'Cancel' button and a blue 'Create New App' button.

Once you've submitted the form, you'll be taken to a dashboard, and under the `Your Apps` section, you should see your newly created app with an API Key that you will use for this project. As a reminder, API keys normally shouldn't be stored in client-side JavaScript. You would normally want to store the keys in your server-side code. To keep this project a simple, front-end only project, you'll store the API key in a front-end environment variable for convenience. Take a moment to create an `.env` file in the root of your project and set an environment variable with your API key, like so:

```
REACT_APP_GIPHY_API_KEY=<<YOUR API KEY>>
```

Notice that your `config.js` file is already exporting your `REACT_APP_GIPHY_API_KEY` environment variable as `apiKey`:

```
export const apiKey = process.env.REACT_APP_GIPHY_API_KEY
```

This means that you can import the API key from any of your frontend components with the following import statement:

```
import { apiKey } from '../config';
```

Phase 1: Fetch data in the Redux cycle

Before you begin to build the project, it's important to think about the state shape. You know that you want to display GIF results returned by a fetch request. This means you'll probably want a `gifs` slice of the state that holds a collection of `gif` objects.

State shape

```
{
  gifs: [
    // gif objects
  ]
}
```

As a reminder, you pass the `gifs` slice of state as a prop to the `App` component in the `AppContainer` through the `mapStateToProps` function.

API util

The first part of creating your Redux cycle for fetching GIFs is creating a fetch request that will be connected to a *thunk action creator*. Define and

export a `fetchGifs` function in the `apiUtil.js` file. This function will make a fetch call to the Giphy API's search endpoint.

```
// apiUtil.js
import { apiKey } from '../config';

export const fetchGifs = searchTerm => (
  // TODO: Write a fetch call to the Giphy API's search endpoint
)
```

It will take a single argument, the `searchQuery` entered by a user. You can check out the [Giphy API docs](#) for more details, but in short, you want to make a fetch request to the following endpoint:

```
`http://api.giphy.com/v1/gifs/search?api_key=${apiKey}&q=${searchTerm}&limit=3`
```

The `searchQuery` will be replaced with your user's actual query. You should tag `&limit=3` onto the end of your query string to tell Giphy you only want three GIF responses. The Giphy API is relatively slow, so keeping the response size down helps optimize your application's performance.

Remember, it's best to test small pieces as we go. Let's test out that fetch request from your developer tools console to make sure it's doing what you're intending.

You may need to restart your server to have your application process the environment variables set in your `.env` file.

Import your `fetchGifs` function to the entry `index.js` file, then go ahead and put it on the window so we have access to it in the console:

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import Root from './components/Root';
import { fetchGifs } from './util/apiUtil';

window.fetchGifs = fetchGifs;

ReactDOM.render(
  <React.StrictMode>
    <Root />
  </React.StrictMode>,
  document.getElementById("root")
);
```

Try running this test code:

```
fetchGifs('puppy').then(res => res.json()).then(res => console.log(res.data));
```

This will make the fetch request, which will return a promise. You'll chain on a `.then` to parse the response and log the parsed response data. You should see an array of three objects. Those are your gif objects fetched from the Giphy API! Make sure you get this working before moving on, and don't forget to remove `fetchGifs` from the window once you're done testing.

Phase 2: Redux actions

Next, you'll set up an [action](#) to properly receive GIF payload information (your fetch responses). As always, you want to export constants for your action types set to strings of your action types. As a reminder, this is to prevent bugs from mistyping your action types.

```
export const RECEIVE_GIFS = 'RECEIVE_GIFS';
```

Now it's time to write a function that returns your `action`, an object literal. Write `receiveGifs` as a function that takes in `gifs` data as a parameter and returns an action object. The object should have two keys: one for the `type` and another for the `gifs` data. Your function should look like the following:

```
const receiveGifs = gifs => {
  return {
    type: RECEIVE_GIFS,
    gifs
  }
};
```

Before testing this action creator, you'll need a reducer.

gifsReducer

Let's write a switch case and default switch return in your `gifsReducer.js` file. Note that the `gifsReducer` function receives the previous `state` and an `action`. Recall that a reducer describes how a slice of state should change based on a dispatched action. It should always return the new state without mutating the previous state. If the action dispatched to the reducer should not change the state, the reducer should return the previous `state` by default. You will need to import the `RECEIVE_GIFS` constant from your `gifActions.js` file.

Your reducer should look similar to the this one:

```
// TODO: Import the `RECEIVE_GIFS` constant

const gifsReducer = (state = [], action) => {
  switch (action.type) {
    // TODO: Return the GIFs from the action object if the action type is `RECEIVE_GIFS`
    // TODO: Return the previous state by default
  }
};

export default gifsReducer;
```

rootReducer

Recall the state shape you saw earlier in the project instructions. The `gifsReducer` above should control the `gifs` slice of the application state. You'll create and export a `rootReducer` with Redux's `combineReducers` function to assign control of different slices of state to their prospective reducer functions to create the application state structure.

This project only needs one reducer, but using `combineReducers` would allow you to easily add more state slices in the future.

The `combineReducers` function has already been imported for you. Take a moment to import your `gifsReducer` and set the `gifs` slice of state to its reducer, like so:

```
gifs: gifsReducer,
```

Now that you have your reducers set up to structure your application's global state, you'll need to set up the Redux store to hold that global state.

Phase 3: The Redux store

The store holds the global state of an application, so you'll need to create it before you can test your reducer. Remember that Redux provides a `createStore` function that receives a `reducer`, optional `preloadedState`, and an optional `enhancer`. Begin by writing a `configureStore` function that passes your `rootReducer` to `createStore`.

```
// store.js
import { createStore } from 'redux';
// TODO: Import middleware
import rootReducer from './reducers/rootReducer';

const configureStore = () => {
  return createStore(rootReducer);
};

export default configureStore;
```

Import `configureStore` into your entry `index.js` file, then use the function to generate the Redux store.

```
const store = configureStore();
```

Now you'll work on providing the store you have generated to your application's components! Begin by passing the `store` you've generated as a prop to the `Root` component. Your `index.js` file should look something like this:

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import Root from './components/Root';
import configureStore from './store';
import { fetchGifs } from './util/apiUtil';

window.fetchGifs = fetchGifs;
const store = configureStore();

ReactDOM.render(
  <React.StrictMode>
    <Root store={store} />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Now go into your `Root.js` file and import `Provider` from `react-redux`. Use the `Provider` component to set the Redux store through the `Provider` component's expected `store` prop. Remember how you have just configured and passed a `store` as a prop to the `Root` component in your entry file (`index.js`). Use the `Root` component's `store` prop to set the `store` prop of the `Provider`:

```
// Root.js
import React from 'react';
import { Provider } from 'react-redux';
import App from './App';

const Root = ({ store }) => (
  <Provider store={store}>
    <App />
  </Provider>
);

export default Root;
```

Now that you have configured your Redux store and tested your `fetchGifs` function, let's connect the function to the store to implement a full Redux cycle. In the next phase, you'll define a *thunk action creator* function and use the `thunk` middleware so that each fetch response is dispatched as a change to the global application `state`.

Phase 4: Thunk middleware

Let's refactor how you fetch GIFs by using a thunk action creator. Recall that we use a *thunk action creator* to return a function. When that function is called with an argument of `dispatch`, the function can dispatch additional actions.

Begin by refactoring your `configureStore` function in the `store.js` file to incorporate your thunk middleware. Remember that Redux provides `thunk` middleware from the `redux-thunk` module. Import the `thunk` middleware and `applyMiddleware` function from Redux. You'll use the `applyMiddleware` function, with your `thunk` middleware as an argument, to set the optional `enhancer` argument in the `createStore` function:

Your `store.js` file should now include the following:

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers/rootReducer';

const configureStore = () => {
  return createStore(rootReducer, applyMiddleware(thunk));
};

export default configureStore;
```

Now your Redux cycle is almost good to go! To summarize your progress to this point, you have configured your Redux store, provided the store to your

application, written a fetch function in a util file, defined an action type, and defined an action creator.

The last step is to define a *thunk action creator* that will dispatch the `receiveGifs` action after the Giphy API call is successful!

Begin by importing your API util function into your `gifActions.js` file. Since you want to easily associate your *thunk action creators* with a specific action, and you'll often have more than one function exported from your util file, use a [namespace](#) to import your fetch function:

```
import * as APIUtil from '../util/apiUtil';
```

You would then invoke your `fetchGifs` function in your `apiUtil.js` file like so:

```
APIUtil.fetchGifs(searchQuery);
```

Now it's time to write your *thunk action creator*! Define and export a function named `fetchGifs` that receives a search term and returns a function that can be called with `dispatch`. Your function will use a promise to parse the fetch response to JSON and dispatch the `receiveGifs` action with the fetch response data after the `APIUtil.fetchGifs` call is successful.

Your *thunk action creator* should look like the following:

```
export const fetchGifs = searchTerm => {  
  return dispatch => {  
    return APIUtil.fetchGifs(searchTerm)  
      .then(res => res.json())  
      .then(res => dispatch(receiveGifs(res.data)));  
  }  
};
```

Or you can clean up the function by using implicit returns with ES6 arrow functions:

```
export const fetchGifs = searchTerm => dispatch => (  
  APIUtil.fetchGifs(searchTerm)  
    .then(res => res.json())  
    .then(res => dispatch(receiveGifs(res.data)))  
);
```

Phase 5: Test your thunk action creator

Let's take a moment to test your the `fetchGifs` *thunk action creator* you just defined.

Import `fetchGifs` from your `gifActions.js` file to your entry `index.js` file. You'll hit the error `Parsing error: Identifier 'fetchGifs' has already been declared` since have already imported the `fetchGifs` function from your `apiUtil.js` file. Note that this is why using a namespace to `import * as APIUtil` from your `apiUtil.js` file is important.

Update the import statement from your `./util/apiUtil.js` file with a namespace:

```
import * as APIUtil from '../util/apiUtil';
```

Now you'll want to put your fetch function, thunk action creator, and Redux store on the window. This way you can view your application's global state and compare the result of your fetch function and the dispatched thunk action creator:

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import Root from './components/Root';
import configureStore from './store';
import * as APIUtil from './util/apiUtil';
import { fetchGifs } from './actions/gifActions';

const store = configureStore();
window.apiFetchGifs = APIUtil.fetchGifs;
window.fetchGifs = fetchGifs;
window.store = store;

ReactDOM.render(
  <React.StrictMode>
    <Root store={store} />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Now you'll use the `getState()` and `dispatch(action)` [store methods](#) to view and update your application's global state. Try the following code in the browser's console before continuing to the next phase:

```
// Return the initial application state
store.getState();

// Use the thunk action creator to dispatch the fetch response and populate state
store.dispatch(fetchGifs('puppy'));

// Return the application state populated with GIFs
store.getState();
```

Congratulations! You just wrote a Redux cycle to populate your application's global state with a response from the Giphy API.

Notice how your application's global state changed after invoking the `dispatch(fetchGifs('puppy'))` method. Now you can pass the `fetchGifs` *thunk*

action creator through a Redux container so that a fetch call can be dispatched from your component to update the global state in the same way!

Phase 6: Logger middleware

Now instead of manually logging the status of your global state and the response of your dispatch call, you can use the Redux `logger` middleware to automatically do so. Import `logger` from `redux-logger` into your `store.js` file:

```
import logger from 'redux-logger';
```

Just like how you applied your `thunk` middleware to your configured store, you'll invoke the `applyMiddleware()` function with your `logger` middleware. Your updated `store.js` file should look something like this:

```
// store.js
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import logger from 'redux-logger';
import rootReducer from './reducers/rootReducer';

const configureStore = () => {
  return createStore(rootReducer, applyMiddleware(thunk, logger));
};

export default configureStore;
```

Now go back to your browser's console and dispatch a fetch request:

```
store.dispatch(fetchGifs('puppy'))
```

Upon a successful fetch, you should have received a helpful log with your application's `prev state`, the dispatched `action` object, and your

application's `next state` from the `logger` middleware you just utilized!

Phase 7: Container component

Now that you've set up the global state of your application, it's time to connect your React components to the global state! Remember how you used the `Provider` component in your `Root.js` file to pass the configured `store` as a prop to the `Provider` component.

Just like how a `Context.Provider` component expects `value` as a prop to set the context object, the Redux `Provider` component expects `store` as a prop to share the store's global state with nested components. Instead of rendering your `App` component as the child of the `Provider`, you'll render an `AppContainer`.

AppContainer

In your `AppContainer.js` file, you will define a `mapStateToProps` function and a `mapDispatchToProps` function and invoke the `connect()` function from Redux. Invoking the `connect()` function will connect the `App` component with the Redux store. Within the container, you will pass slices of `state` and dispatched *thunk action creators* as props to the `App` component.

Note that the skeleton has already imported the `connect` function from `react-redux`, your `fetchGifs` thunk action creator from the `gifAction.js` file, and the `App` component.

Take a look at the code skeletons of the `mapStateToProps` and `mapDispatchToProps` functions. Think of these functions as functions that take in `state` or `dispatch` as arguments to return object literals that represent the props that your `App` component will receive.

Have your `mapStateToProps` function return an object with a `gifs` property set to the `state.gifs` slice of state. Doing this will pass a `gifs` prop into your

`App` component.

Next, have your `mapDispatchToProps` function return an object with a `fetchGifs` property set to an arrow function that accepts a `searchQuery` value and dispatches a call to the `fetchGifs(searchQuery)` function. Remember how you tested the `store.dispatch(fetchGifs(searchQuery))` function in the browser console. Think of how you would use an arrow function to almost wait for a `searchQuery` input from the user before firing the dispatch call.

Lastly, take a moment to examine how your `AppContainer.js` file is simply exporting the `connect()` invocation:

```
export default connect(mapStateToProps, mapDispatchToProps)(App);
```

Now you can render your `AppContainer` instead of your `App` component so that the `connect()` method is invoked to pass a `gifs` prop and a `fetchGifs` props to `App`!

Take a moment to refactor your `Root.js` file to do so:

```
// Root.js
import React from 'react';
import { Provider } from 'react-redux';
import AppContainer from './AppContainer';

const Root = ({ store }) => (
  <Provider store={store}>
    <AppContainer />
  </Provider>
);

export default Root;
```

Phase 8: Presentational components

Now that you've created a container to pass (or *map*) slices of the global state and dispatched actions as props, it's time to render and update the `gifs` from the global state!

App

Have your `App` component take in and destructure your `gifs` and `fetchGifs` props. Notice how your component is rendering a `SearchBar` component and a `GifsList` component. You'll want to pass in the `fetchGifs` function as a prop to the `SearchBar` component.

Instead of directly passing all of the `gifs` as a prop to the `GifsList` component, you'll refactor your `mapStateToProps` function and use a selector to map the array of `gifs` into an array of GIF urls.

Currently, your `mapStateToProps` function should look something like this:

```
const mapStateToProps = state => {
  return {
    gifs: state.gifs,
  };
};
```

Define a `getGifUrls` selector that takes in the `state` and uses parameter destructuring to map over each gif in the `gifs` slice of state to pluck the image URL from the JSON data, like so:

```
const getGifUrls = ({ gifs }) => (
  gifs.map(gif => gif.images.fixed_height.url)
);

const mapStateToProps = state => {
  return {
    gifUrls: getGifUrls(state),
  };
};
```

Now you can pass in the `gifUrls` for your `GifsList` component to render GIF images without passing unnecessary data as props!

SearchBar

In your `SearchBar` component, you already have a search form, an `inputValue` state, and an `onChange` handler for the form's input field set up. Right now, your component has several `TODO` notes to guide your creation of an `onSubmit` handler for your search form. The `onSubmit` handler will dispatch the `fetchGifs` action creator function (don't forget to prevent the default action of a submit event).

After finishing your `SearchBar` component, test out your submit event handler and check your `logger` response to see if your `fetchGifs` action creator is actually being dispatched to update the application's global state! Once you see that your fetched `gifs` have been dispatched to the `gifs` slice of state, it's time to render your list of GIFs!

GifsList

Right now, your component has several `TODO` notes:

- Take in and destructure the `gifUrls` prop.
- Render a `<div>` as the parent element of your `GifsList` component.
- Map over your `gifUrls` array to render an `` for each `url`.

Debugging

If you're having issues rendering a GIF in your project, that means it's a great time to practice using `debugger` statements to debug your Redux cycle code! For example, you could set a `debugger` in the:

- `handleSubmit` method to check out the submitted search `inputValue`

- `receiveGifs` action creator function to check out the value of the `gifs` payload
- `RECEIVE_GIFS` case statement in the `gifsReducer` to check out the value of the dispatch `action` object

If you didn't hit any issues, congratulations - you have implemented Redux with the Giphy API to create a search API that renders a list of GIFs! Before moving forward to the next project, you should still use `debugger` statements to step through this project's Redux cycle.

WEEK-15 DAY-4

Hooks

Hooks Objectives

Now that you've learned the basic objectives of using React, you should be able to gain a fundamental understanding of the React, Redux, and React-Router hooks. In your software engineering career, official documentation will be your friend! It's important to learn how to navigate through official documentation. At the end of the readings, you should use your new fundamental understanding of hooks to go through the official React Hooks documentation. At the end of this topic's articles and lectures you should be able to create function components that use state and other React features.

You should be able to use React's:

- `useState` hook to manage a component's state.

- `useState` hook to set a default state, instead of setting the default state in a `constructor()` method.
- `useState` hook to update state, instead of the `setState()` method.
- `useEffect` hook to manage *side effect* operations (i.e. data fetching).
- `useEffect` hook in replacement of commonly used component lifecycle methods (`componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`).
- `useEffect` (and the hook's *dependency array*) to optimize an application's performance by skipping `useEffect` calls.
- `useContext` hook to access a context object, instead of a `Context.Consumer` or the `static contentType` property.

You should be able to use Redux's:

- `useSelector` hook to access the Redux store's state from within a component (instead of passing a part of state as a prop with the `mapStateToProps` function).
- `useDispatch` hook to dispatch an action from within a component (instead of passing an thunk action creator function through the `mapDispatchToProps` function).

You should be able to use React Router's:

- `useParams` hook to match parameters in the current route (instead of accessing the `match.params` prop).
 - `useHistory` hook to navigation from within code (without `<Link>`, `<NavLink>`, or the `history` prop).
 - `useLocation` hook to track url changes.
 - `useRouteMatch` hook to check if the current url matches a path format.
-

Intro to React Hooks

React Hooks are a way for function components to have the same functionality as class components that make use of component lifecycle methods. Hooks are simply functions that allow components to utilize React features without explicitly using the lifecycle methods.

Before React Hooks, the only way to use lifecycle methods were through class components. Hooks allow you to manage a component's state and lifecycle within function components. They are helpful in extracting stateful logic from a component to be independently tested and reused - it's much more complicated to test the functionality of logic in a component's lifecycle methods. After reading this article, you will:

- Have a general understanding of the features of basic React hooks
- Understand how the basic Hooks connect to features of React class components (i.e. lifecycle methods)
- Create function components that use state and other React features
- Use the `useState` hook to manage a component's state
- Use the `useEffect` hook to manage *side effect* operations (i.e. data fetching)
- Use the `useContext` hook to access a context object

useState

Up to this point, you have set a component's default state within a component's `constructor` method. The `useState` hook replaces the need to use a constructor to declare a default state with `this.state`. You can use the `useState` hook to set and name a default slice of state without a `constructor()` method. You can set a default state simply by invoking the `useState` hook. The **with hooks** example below sets the default `inputValue` state to be `'Default input value here!'` by invoking the `useState` hook with the string `'Default input value here!'`.

with hooks

```
const FormWithHooks = () => {  
  const [inputValue, setInputValue] = useState('Default input value here!');  
};
```

without hooks

```
class FormWithoutHooks extends React.Component {  
  constructor() {  
    super();  
    this.state = {  
      inputValue: 'Default input value here!'  
    };  
  }  
}
```

When you use the `useState` hook to set up a slice of state, you also set up a prospective function to update that slice of state. In this example, you can update a slice of state by invoking `setInputValue`, instead of invoking the `this.setState()` method.

with hooks

```
const updateInputVal = e => setInputValue(e.target.value);
```

without hooks

```
updateInputVal = e => this.setState({ inputValue: e.target.value });
```

In general, React Hooks help clean up your code **a lot!** For example, when using the `useState` hook, you can also simply reference `inputValue` throughout the component, instead of `this.state.inputValue`. Compare the difference between the code for the `FormWithHooks` and `FormWithoutHooks` components below.

with hooks

```
const FormWithHooks = () => {
  const [inputValue, setInputValue] = useState('');
  const updateInputVal = e => setInputValue(e.target.value);

  return (
    <form>
      <input
        type="text"
        value={inputValue}
        onChange={updateInputVal}
        placeholder="Type something!"
      />
    </form>
  );
};
```

without hooks

```
class FormWithoutHooks extends React.Component {
  constructor() {
    super();
    this.state = {
      inputValue: '',
    };
  }

  updateInputVal = e => this.setState({ inputValue: e.target.value });

  render() {
    return (
      <form>
        <input
          type="text"
          value={this.state.inputValue}
          onChange={this.updateInputVal}
          placeholder="Type something!"
        />
      </form>
    );
  }
}
```

When refactoring your projects to implement React Hooks, you can always refactor component by component, starting out with refactoring the component's state management.

useEffect

The `useEffect` hook is used to manage side effect operations. An example of a side effect operation you are familiar with is data fetching. Similarly to the `componentDidMount` or `componentDidUpdate` lifecycle methods, the `useEffect` hook will automatically run.

Take a moment to notice how using the `useEffect` hook simply means invoking the `useEffect` function. You can invoke the function with one or two arguments,

with the first argument always being a function, and then second argument being an optional *dependency array*.

When the `useEffect` hook is invoked **without** a second argument, the function will be invoked after every render:

```
useEffect(() => {  
  // Side effect logic invoked after every render  
});
```

When the `useEffect` hook is invoked **with an empty array**, the function is only invoked once, when a component mounts (think of `componentDidMount`):

```
useEffect(() => {  
  // Side effect logic invoked once, when a component mounts  
}, []);
```

When the `useEffect` hook is invoked **with an array of dependencies**, the function is invoked whenever a dependency changes (think of `componentDidUpdate`):

```
useEffect(() => {  
  // Side effect logic invoked every time the `dependentVariable` changes  
}, [dependentVariable]);
```

Skipping effects with the dependency array

This second argument of the `useEffect` hook is known as the *dependency array*. You can optimize the performance for your component by using the dependency array to skip effects. The dependency array is a collection of dependent variables. Similarly to how the `componentDidUpdate` lifecycle method listens for a change in the component, the `useEffect` hook listen for changes to variables in the dependency array to determine whether or not to run the *effect* again.

```
useEffect(() => {  
  // Side effect logic  
}, [/* Dependency array */]);
```

Asynchronous effects

You are familiar with using `async/await` to await a database fetch. If you'd like to make an asynchronous fetch within a `useEffect` hook, you would declare an asynchronous function within the hook. Then, you would invoke the asynchronous functions from within the hook.

```
useEffect(() => {  
  const fetchSomething = async () => {  
    // Fetch call  
  };  
  
  fetchSomething();  
}, [/* Dependency array */]);
```

The function passed in as the `useEffect` hook's first argument **cannot** be an asynchronous function - this is why you need to define and invoke the asynchronous function from within the hook's first function argument.

In the example below, the `useEffect` hooks runs an asynchronous fetch of a puppy, based on a `puppyId` input. The hook's dependency array references `props.match.params.puppyId`. Since the `useEffect` hook's dependency array references the `puppyId` parameter, the application will only fetch whenever the `puppyId` parameter changes. This optimizes the code, because now the effect is only run upon the change of a specific variable - `puppyId` !

```
useEffect(() => {
  const fetchPuppy = async (puppyId) => {
    const puppy = await fetch(`https://api.puppies.example/${puppyId}`);
    const puppyJSON = await res.json();
    return puppyJSON;
  };

  fetchData(props.match.params.puppyId);
}, [props.match.params.puppyId]);
```

Using a dependency array also prevents endless loops. Without the dependency array, a fetch call invoked within a `useEffect` hook would constantly run and your code would error out.

Alternatively, you can invoke the asynchronous effect with an `IIFE` (immediately invoked function expression). Take the example syntax below:

```
useEffect(() => {
  (async function fetchSomething() {
    // Fetch call
  })();
}, [/* Dependency array */]);
```

Effect cleanup

In a class component, you might use the `componentWillUnmount` lifecycle method to handle *cleanup*. In order to *cleanup* an effect, you would need to return a function from within the `useEffect` hook. Having the `useEffect` hook's callback return another function results in the cleanup behavior of `componentWillUnmount`.

```
useEffect(() => {
  return function cleanup() {
    // Cleanup logic
  }
}, [/* Dependency array */]);
```

In a later lesson, you will learn about how to use WebSockets. When you use a WebSocket, you create a connection. What if you want to close that connection? Closing a connection sounds like a *cleanup* task! It is common to invoke the WebSocket's `close` method in a *cleanup* function. The example below makes use of the *dependency array* and a *cleanup* function.

```
useEffect(() => {
  if (!username) {
    return;
  }

  const ws = new WebSocket('ws://localhost:8080');
  websocket.current = ws;

  return function cleanup() {
    if (websocket.current !== null) {
      websocket.current.close();
    }
  };
}, [username]);
```

Similar to the behavior of `componentDidUpdate`, the effect is re-run whenever the `username` changes. The `useEffect` hook below takes care of setting up a new WebSocket connection. The hook's *cleanup* function will be run whenever the component unmounts. Replacing the `componentWillUnmount` lifecycle method, the *cleanup* function will take care of closing the WebSocket connection when the component unmounts.

useContext

You can use the `useContext` hook to access a context object to read and subscribe to context changes. The `useContext` hooks replaces the `static contextType` property in class components. Whenever you used the `static contextType` property in a class component, you were able to access a context object via referencing `this.context`. When you use the `useContext` hook, you

can access a context object via whatever you name the context! In the example below, the `useContext` hook is invoked and its return value (the `MyContext` object) is named `context` - this means you can access the `MyContext` object anywhere within the component via referencing `context` .

with hooks

```
const context = useContext(MyContext); // Makes `MyContext` available as `context`
const banana = useContext(BananaContext); // Makes `BananaContext` available as `banana`
const puppy = useContext(PuppyContext); // Makes `PuppyContext` available as `puppy`
```

without hooks

```
static contextType = MyContext; // Makes `MyContext` available as `this.context`
```

When using the `useContext` hook to access a context object, you would still use a `<Context.Provider>` to set the context's `value` .

What you have learned

In this article, you have learned about the general features of the basic React hooks (`useState` , `useEffect` , and `useContext`). You should now understand the functionality of how the basic Hooks connect to the features of React class components. You should be able to use the:

- `useState` hook to manage a function component's state
 - `useEffect` hook to manage running, skipping, and cleaning up effects
 - `useContext` hook to access a context object
-

Introduction to Hooks in Redux

In previous lessons and projects, you have learned to build **React** components using **Redux**. Now it's time to explore ways to modify your approach using hooks.

When you complete this lesson, you will be able to

- Use Redux in a function component with the `useSelector` and `useDispatch` hooks

Using hooks with Redux

In order to use hooks in Redux, your application will need to utilize the `react-redux` package. If you need a refresher on what this kind of application looks like, see the [Starting Point](#) section at the end of this reading or clone the [intro-to-redux-hooks](#) repository from GitHub and look at the **starter** folder.

Getting Started

Consider a simple application that displays the user's current IP Address with a button to start the lookup. You may even include a loading message which shows while the server call is running.

```
// ./src/App.js

import React, { useEffect, useState } from 'react';

const App = props => {
  const [ip, setIP] = useState(null);
  const [loading, setLoading] = useState(false);

  const getMyIP = () => {
    setIP('(coming soon)');
  };

  useEffect(() => {
    setLoading(ip === "");
  }, [ip]);

  return (
    <div>
      <h1>Get My IP</h1>

      {loading
        ? <p>Loading...</p>
        : <p>{ip}</p>
      }
      <button
        onClick={getMyIP}
        disabled={loading}
      >{ip ? 'Again' : 'Go'}</button>
    </div>
  );
};

export default App;
```

Notice that this framework uses your knowledge of the `useState` hook to simulate the server call and the `useEffect` hook to cause the loading indicator to show at the appropriate times.

Now you can update this example to use Redux hooks to replace the fake loading of `ip`.

useSelector

Begin by importing `useSelector` from the *React Redux* package.

```
import { useSelector } from 'react-redux';
```

Assuming you have a reducer with the property `ipAddress`, then you can use the `useSelector` hook to access the `ipAddress` from your Redux store's state.

```
const ip = useSelector(state => state.ipAddress);
```

In the sample *App.js* above, using the `useSelector` hook would replace `const [ip, setIP] = useState('')`. Your component would receive the `ip` via your Redux store's `state.ipAddress`, instead of the component's `ip` state.

As a reminder, the `useState` hook in this example is simply mimicking a fetch response. Upon clicking the button with the `getMyIP` click handler, a fetch call is mimicked with the `setIP('(coming soon)')` method. You will need to remove this line as well. Don't worry you'll replace it momentarily using another Redux hook.

You can access any available property this way and even call `useSelector()` multiple times within a single function component. You can even use props or route parameters to determine what to extract from the store.

Here is an example using props. Assume you have a store with a `users` object in its state. Furthermore, you want to get just `user` based on the `id` provided in a prop to a function component.

Here is the component's code. See if you can spot where the "magic" happens.


```
import React from "react";
import { useSelector } from "react-redux";

const UserCard = props => {
  const user = useSelector(state => state.users[props.id]);
  return <div>{todo.text}</div>;
};

export default UserCard;
```

If you said the magic happens in the function passed to the `useSelector` hook, then you would be correct. Specifically the square bracket notation is used to get just a part of the `users` object. Remember, you're passing a function as the argument to `useSelector`; therefore you can use all your skills to determine the right object or value to `return`.

useDispatch

In order to trigger an action in **Redux**, you will need to utilize a different hook; specifically, `useDispatch()`. This hook returns a function which you can call to dispatch the action.

```
const dispatch = useDispatch();
```

Exactly how you use dispatch depends on your Redux setup. There are some minor differences based on whether you decided to use `redux-thunk` in your project. The configuration of Redux is beyond the scope of this reading and is something you saw in previous activities. Two solutions are provided in the sample, so you can make the choice which works best for your project. Here's a quick look at these two options.

Option A: Generic Redux (no thunk)

In this configuration, you will need to dispatch actions created in your Redux component (e.g. *src/store/ipAddress.js*). For example, one possible action creator function might look like

```
export const setIP = ip => ({ type: SET_IP, ip });
```

Any functions which perform loading operations will need to be asynchronous and return the value or object retrieved; perhaps in a scenario like this...

```
// relevant snippet from of src/store/ipAddress.js

export const loadIP = async () => {
  // ...
  // do stuff here like a fetch with await
  // ...
  // return the result
  return origin
};
```

Back in the component with the UI (e.g. *src/App.js*), you'll need to start by importing these functions as well as adding `useDispatch` to the import for **Redux**.

```
import { useDispatch, useSelector } from "react-redux";
import { loadIP, setIP } from "../store/ipAddress";
```

Then use these with your button click handler. Notice you dispatch is using the action to set the value of the ip variable that you just got with `useSelector`.

```
// relevant snippet from src/App.js
```

```
const dispatch = useDispatch();
```

```
const getMyIP = async () => {  
  dispatch(setIP(""));
```

```
  const origin = await loadIP();  
  dispatch(setIP(origin));  
};
```

The example dispatches two values for the IP Address. The first dispatch call sets the IP address to an empty string (so that the old value no longer shows in the UI while the newer value is loading). The second, of course, is the result of the fetch (or any other kind of service call, of course).

One advantage of this approach is that you will not need to install `redux-thunk` or add it to the **Redux** configuration. However, this comes with the trade-off that actions will be dispatched throughout the application, including in UI components.

Option B: Using Redux Thunk

Now consider the difference using `redux-thunk`. The action function remains unchanged (`export const setIP = ip => ({ type: SET_IP, ip });`). The `loadIP` function will do its own dispatching (this means a double function in the declaration that results in the code below).

```
// relevant snippet from ./src/App.js
```

```
export const loadIP = () => async dispatch => {  
  dispatch(setIP(""));
```

```
  // ...  
  // do stuff here like a fetch with await  
  // ...  
  // dispatch the result  
  dispatch(setIP(origin));  
};
```

In the component (e.g. `_src/App.js`), you'll need to import only the `loadIP()` function (and not the `setIP` action creator function) (while still importing `useDispatch`, of course).

```
import { useDispatch, useSelector } from "react-redux";  
import { loadIP } from "../store/ipAddress";
```

Then the click handler for the button simplifies to

```
// relevant snippet from ./src/App.js
```

```
const getMyIP = () => {  
  dispatch(loadIP());  
};
```

The advantages of this approach using **Redux Thunk** is the separation of responsibilities where the load and action dispatches are all together resulting in simplified handling within the UI components.

The trade-off is double functions in your Redux (like `export const loadIP = () => async dispatch => { }`) and the one-time install and setup of `redux-thunk`.

Ultimately the decision on the approach is made by each development team based on their personal preference.

Refactoring an existing component

In order to refactor an existing class component from the classic approach to using hooks, there are several steps that need to be taken:

- Change component definition from class to function
 - Switch state handling to the `useState` hook
 - Use the `useEffect` hook for side-effect management, instead of the `componentDidMount` and `componentDidUpdate` methods
 - Create or move event handlers to constant functions
- Use one or multiple `useSelector` hooks to replace the `mapStateToProps` function
- Invoke the `useDispatch` hook to use `dispatch` and replace the `mapDispatchToProps` function
- Simplify the `export` to just the component name by removing `connect`
- Delete any imports that are no longer in use

The best way to understand exactly what to do is to see an example. This will be provided in an upcoming video lesson.

What you have learned

The `react-redux` package comes with several hooks which can be used to replace `mapStateToProps`, `mapDispatchToProps` and `connect`. Hooks are used with function components, so remember to start with one if you intend to use hooks; otherwise you'll need to convert your class component to a function component.

The `useSelector` hook give you access to any and all props that are exposed through the state in a Redux store by passing in a function to resolve the

state property you want (e.g. `useSelector(state => state.theProp)`). The `useDispatch` hook allows you to trigger an action directly or by calling a function that uses `redux-thunk` to dispatch the action.

Using hooks with **React Redux** can improve the readability and maintainability of a **React** project.

Additional resources

For future reference, there are a few additional (advanced and rarely used) features in the [official documentation on hooks in React Redux](#).

Starting point for IP address project

As promised, here is an example of setting up the framework with Redux for the "Get My IP" application discussed throughout this reading. This version includes **Redux Thunk**.

You may access the starter project, the solution project with Redux Thunk, and the solution project without Redux Thunk by cloning the [intro-to-redux-hooks](#) repository.

Start with `create-react-app` and install `react-redux`, `redux-thunk` and their dependencies (e.g. `redux`) as you've done previously.

Wrap your application in the **Redux Provider** ...

```
// ./src/index.js

import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import App from './App';
import configureStore from './store/configureStore';

const store = configureStore();

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
);
```

... and configure a **Redux** store ...

```
// ./src/store/configureStore.js

import { createStore, applyMiddleware, combineReducers, compose } from 'redux';
import thunk from 'redux-thunk';
import ipAddress from './ipAddress';

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

const reducer = combineReducers({
  ipAddress,
});

const configureStore = initialState => {
  return createStore(
    reducer,
    initialState,
    composeEnhancers(applyMiddleware(thunk)),
  );
};

export default configureStore;
```

... which includes a **reducer**, an **action creator** function, and a **thunk action creator** function ...

```
// ./src/store/ipAddress.js

import { ipUrl } from '../config';

const SET_IP = 'ipAddress/SET_IP';

export const setIP = ip => ({ type: SET_IP, ip });

export const loadIP = () => async dispatch => {
  dispatch(setIP(""));

  const response = await fetch(`${ipUrl}/ip`, {
    method: 'get',
    headers: { 'Content-Type': 'application/json' },
  });

  if (response.ok) {
    let { origin } = await response.json();
    // obscure last segment for privacy purposes
    origin = origin.split('.', 3).join('.') + ".xxx";
    // dispatch the result
    dispatch(setIP(origin));
  }
};

export default function reducer(state = {}, action) {
  switch (action.type) {
    case SET_IP: {
      return {
        ...state,
        ip: action.ip,
      };
    }

    default: return state;
  }
}
```

... that relies on the application configuration ...

```
// ./src/config.js

export const ipUrl = process.env.REACT_APP_BASEURL || `https://httpbin.org`;
```

... to fetch the IP Address using the [ip query at httpbin.org](https://httpbin.org/ip).

React Router Hooks

Now it's time to dig into the specifics of how hooks can simplify **React** code when working with React Router, specifically `react-router-dom`.

When you complete this lesson, you should be able to use the hooks that are built into the `react-router-dom` package:

- `useParams` for matching parameters in the current route
- `useHistory` for navigation from code (without `Link` or `NavLink`)
- `useLocation` for tracking url changes
- `useRouteMatch` for checking if the current url matches a path format

useParams

The most common usage of hooks with `react-router-dom` is the case where a RESTful path has one or more parameters, such as an `id`.

For example, the `id` in a path like `/user/:id` may be accessed as the property of an object returned by `useParams()`.

Option 1

```
const params = useParams();
console.log('User id is', params.id);
```

Option 2 (more common)

```
const { id } = useParams();
console.log('User id is', id);
```

Now, consider this path `/user/:userId/doc/:docId` . It has two parameters, `userId` and `docId` ; therefore, they would be accessed using `const { userId, docId } = useParams()` . Notice how the variables in the path match the properties on the objects returned by `useParams()` .

Here's an expanded example showing a basic function component.

```
// ./src/components/Document.js

import React from 'react';
import { useParams } from 'react-router-dom';

const Document = () => {
  const { userId, docId } = useParams();

  return (
    <>
      <h2>Document {docId}</h2>
      <p>Created by User {userId}</p>
    </>
  );
};

export default Document;
```

As a reminder, you'll need to wrap your components within a `<Router>` in order to use the hooks built into the `react-router-dom` package. Perhaps like this...

```
// ./src/index.js

import React from 'react';
import ReactDOM from 'react-dom';
import {
  BrowserRouter as Router,
  Switch,
  Route,
} from 'react-router-dom';
import Document from './components/Document';

// For simplicity, Router and Switch are here instead of the traditional App.js
ReactDOM.render(
  <Router>
    <Switch>
      <Route path='/user/:userId/doc/:docId' component={Document} />
      { /* Other routes also */ }
    </Switch>
  </Router>,
  document.getElementById('root')
);
```

useHistory

The `useHistory()` hook gives you access to the **history** object, which is a record of paths visited on the current browser tab.

While there are a number of possibilities for what you can do with `history` , some are more useful than others. Here are the top methods and property.

- `push(path, [state])`
 - Adds a new path to the history and navigates there
 - `state` object is optional
- `replace(path, [state])`
 - Removes the current path from history before adding the new path and navigating there
 - `state` object is optional

- When the user goes back from the next path, they will skip the replaced path (either with the browser's BACK button or the `goBack()` function)
- `goBack()`
 - Returns to the previous path in the history
- `location` - the current location
 - `pathname` - the path
 - `search` - query params (following a `?` in the url), if any
 - `hash` - value following a `#` in the url, if any
 - `state` - object provided with `push()` or `replace()`

The `state` object is a way for you to pass one or more data values between routes. The sender creates the object and passes it as the second argument to `history.push` or `history.replace`; the receiver accesses the object using `history.location.state`.

For more capabilities, you can read the [documentation on History](#), if you so desire.

Here's an example of a function component using history for custom navigation.

```
// ./src/components/ComingSoon.js

import React, { useEffect } from 'react'
import { useHistory } from 'react-router-dom'

const ComingSoon = () => {
  const history = useHistory();

  useEffect(() => {
    const tid = setTimeout(() => {
      history.replace('/');
    }, 2000);
    return () => clearTimeout(tid);
  });

  return (
    <h2>Coming Soon</h2>
  );
};

export default ComingSoon;
```

ASIDE: This example also makes use of the `useEffect` hook discussed in other lessons in order to automatically redirect the user after a timeout period. In particular, notice how `return` is used to prevent warnings in React if the user chooses to leave the page before the timeout period ends.

useLocation

The preferred approach to accessing the location from within a component is through the history object. However, there is a special case where the `useLocation` hook is useful - connecting to a service which tracks page loads.

One example is **Google Analytics**.

```
// ./src/index.js

import React from 'react';
import ReactDOM from 'react-dom';
import {
  BrowserRouter as Router,
  Switch,
  useLocation
} from 'react-router-dom';
import ga from 'react-ga';

const TrackingWrapper = ({ children }) => {
  const location = useLocation();
  React.useEffect(() => {
    ga.send(['pageview', location.pathname]);
  }, [location]);
  return children;
}

ReactDOM.render(
  <Router>
    <TrackingWrapper>
      <Switch>
        { /* App and/or Routes, etc. */ }
      </Switch>
    </TrackingWrapper>
  </Router>,
  document.getElementById('root')
);
```

The setup and usage of Google Analytics is beyond the scope of this lesson. However, if you'd like to learn more you can search online for examples, such as [Google Analytics with React](#). In short, the call to `ga.send()` logs whatever event you pass it into your GA account. Then you can sign in to GA to view and analyze the recorded data including days and times when users are most active, what country your visitors are coming from, and much more.

useRouteMatch

If you'd like to check for a matching path before rendering a route, then turn to `useRouteMatch`. This hook accepts an argument which is compared to the current path in the same fashion as `Route` and returns a boolean (`true` or `false`).

For example, `useRouteMatch('/report/advanced')` could be used to show (or hide, when not matching) an advanced user interface for modifying a report on the fly.

Bring it together

Here is an example of a component which lays the framework for a thorough usage of React Router hooks (except `location` which is better used elsewhere). Use your detective skills to figure out as much as you can. A thorough explanation is provided in one of the video lessons.

Imagine the following `Report` component is placed in a router with `<Route path={['/report/:date', '/report']} component={Report}/>`.

BONUS: In case you didn't know already, a router can use an array to specify multiple paths to match with the provided component.


```
// ./src/components/Report.js

import React, {useEffect} from 'react';
import { useRouteMatch, useHistory, useParams } from 'react-router-dom';

const Report = () => {
  const matchUC = useRouteMatch({
    path: '/REPORT*',
    strict: true,
    sensitive: true
  });
  const matchAdvanced = useRouteMatch([
    '/report/advanced',
    '/report/*/advanced'
  ]);
  const matchAll = useRouteMatch('/report/all');
  const { date } = useParams();
  const history = useHistory();

  useEffect(() => {
    if (matchUC)
      history.replace(history.location.pathname.toLowerCase());
  }, [matchUC, history])

  if (matchUC)
    return ""

  if (!date) return (
    <p>Select Report
      <br/><button onClick={() =>
        history.push('/report/last-week')}
      >Last Week</button>
      <br/><button onClick={() =>
        history.push('/report/last-month')}
      >Last Month</button>
      <br/><button onClick={() =>
        history.push('/report/all')}
      >View All</button>
    </p>
  );

  if (date === 'advanced') return (
```

```
    <p>Select Report
      <br/><button onClick={() =>
        history.push('/report/last-week/'+date)
      >Last Week</button>
      <br/><button onClick={() =>
        history.push('/report/last-month/'+date)
      >Last Month</button>
      <br/><button onClick={() =>
        history.push('/report/all/'+date)
      >View All</button>
    </p>
  );

  if (matchAll) return (
    <>
      <h2>Complete Report of Everything</h2>
      {matchAdvanced && <p>... Alternate Advanced Controls ...</p>}
    </>
  )

  return (
    <>
      <h2>Report For {date}</h2>
      {matchAdvanced && <p>... Advanced Controls ...</p>}
    </>
  );
};

export default Report;
```

The various routes to explore include

- `/REPORT` or `/REPORT/SOMETHING/ADVANCED` or any other variation starting with `REPORT` in all caps will redirect to the same url in lowercase
- `/report` or `/report/advanced` will show a few buttons
- `/report/all` will show a different title than `/report/something-else` (with or without the next option)
- any url ending in `/advanced` will show "Advanced Controls"

What you've learned

The `react-router-dom` package comes with hooks you can use to simplify the code in your **React** applications. For example, utilizing

`const { id } = useParams()` within a component displayed in the path `/user/:id` give you access to the value that replaces the `:id` parameter.

Navigating can be accomplished with `const history = useHistory()` followed by `history.push('/a/new/path')`, and you can even include a `state` object as a second parameter. Additionally, `useLocation` can help you connect to tracking services, and `useRouteMatch` might come in handy once in while for pattern matching on the path itself. In short, the handling of RESTful paths in React is enhanced when you embrace the hooks available in React Router.

For future reference, you may want to bookmark the [official documentation on React Router Hooks](#).

React Hooks Documentation

You now know how to manage information in a React application through `state`, `props`, `context`, and a `Redux` store. Take a moment to read through the official React Hooks documentation. Now that you've been working with React for almost two weeks now, you should have a base level understanding of React and the component lifecycle. Congratulations on learning such a concept heavy library!

As you move forward in your development careers, you'll need to lean more on official documentation - React's official documentation is a great place to start! Review the following articles about from the official React Hooks documentation and try to be aware of how you navigate the documentation to aid your own learning.

Why hooks?

- [The Motivation Behind Hooks](#)
- [Hooks at a Glance](#)

Hook basics

- [Rules of Hooks](#)
- [Hooks API Reference](#)
- [useState](#)
- [useEffect](#)
- [useContext](#)
- [Optimizing Performance by Skipping Effects](#)

More about hooks

- [Custom Hooks](#)
 - [Hooks FAQ](#)
-

Pokedex Hooks Project: Phase 1

In today's project you will refactor class-based components that make use of lifecycle methods to function components that make use of React Hooks! At this point, you have already built an application with class components and context to manage application state. You've also built the same application to learn about how to use Redux instead of React Context for state-management.

In this project, you'll build your application with React and Redux hooks! You will implement the:

- `useState` hook to manage a component's state
- `useEffect` hook to manage a component's side effect operations
- `useDispatch` hook to dispatch actions from with a component file
- `useSelector` hook access slices of state from the Redux store
- `useContext` hook to manage your application with Context instead of Redux

Phase 1: State-based hooks application

You'll need the backend for the Pokedex application. Take a moment to clone it from <https://github.com/appacademy-starters/pokedex-backend> and get it set up.

The API for the backend is also documented in repository's README.

Once you have that up and running, you'll begin working out of the solution for the state-based class components project. Begin by cloning the state-based application from

<https://github.com/appacademy-starters/react-hooks-pokedex-starter.git>.

Throughout today, you'll work on refactoring each class component in the application to be a function component that makes use of React Hooks!

Explore the reference application

As you might remember, your current application comprises of the following components:

- `App` : Does the browser routing and top-level fetches of data to draw the data
- `LoginPanel` : Shows the login panel
- `PokemonBrowser` : The browser that draws the list on the left after logging in and has a route to the `PokemonDetail` when the route matches `"/pokemon/:id"`
- `PokemonDetail` : Makes a fetch to the API on mount and update to load the details of the selected Pokemon

Refactor components

As you're refactoring your application's components, you'll most likely hit bugs and break your application. While you're refactoring each component, make sure to test that your refactored code is working before moving on to refactor the next component. As a general guideline, you should refactor each component from the lowest, most nested component up to the top-most parent:

1. `PokemonDetail`
2. `PokemonBrowser`
3. `LoginPanel`
4. `App`

You'll update how each component sets its default state by using the `useState` hook. You'll also refactor the lifecycle methods of each component into side effect operations managed by the `useEffect` hook. At the end of this exercise, you should have a good understanding of how to use the basic `useState` and `useEffect` hooks to write function components with side effect operations.

Take this project as a way to practice learning new technologies by referencing official documentation:

- [Using the State Hook](#)
- [Using the Effect Hook](#)
- [Hooks API Reference](#)

Remember, Create React App will let your React application use environment variables that start with `REACT_APP_`. Just like with your state-based application built with class components, you can import environment variables from the `config.js` file to clean up your code for specifying the URL of the backend.

Once you have finished refactoring, take a moment to commit your changes to the main branch of your `react-hooks-pokedex-starter` project:

```
git add .
git commit -m "Refactor app to implement hooks"
```

In the next two phases, you'll create two different projects in two different branches that use this commit as a starting point. You'll branch off from this point to create a Redux-based project and a Context-based project (both utilizing hooks).

Pokedex Hooks Project: Phase 2

As you might remember from the Redux-based Pokedex project, implementing Redux results in a lot of boilerplate code. Using Redux hooks can help clean up and get rid of a lot of boilerplate code. In this phase you will refactor the Redux-based project to use React hooks and implement Redux hooks!

Begin by creating a new branch for your Redux-based application:

```
git checkout -b redux-hooks-app
```

Take a moment to download the [Redux-based Pokedex hooks starter project](#). Make sure you are in your new `redux-hooks-app` branch:

```
git branch
```

Delete all of your project's current code and move the files of `redux-based-pokedex-solution` into your current directory (`react-hooks-pokedex-starter`). For example, if both project directories are within the same directory, you can use the `mv` command from the two project folders' parent directory.

```
mv -v redux-based-pokedex-solution/* react-hooks-pokedex-starter
```

Now take a moment to commit the start of your Redux-based project:

```
git add .
git commit -m "Initialize redux hooks starter project"
```

Using Redux hooks to manage application state

In this phase, you'll be refactoring all your component files to use Redux hooks instead of the `mapStateToProps`, `mapDispatchToProps`, and Redux `connect` functions. Just like in phase 1, you might hit bugs and break your application while refactoring your application's components. Make sure to test that your refactored code is working before moving on to refactor the next component. As a general overview, you'll be refactoring the code for the following components:

1. `LogoutButton`
2. `LoginPanel`
3. `PokemonDetail`
4. `PokemonForm`
5. `PokemonBrowser`

There are two ways you can refactor your project:

1. You can refactor your project to use the `useDispatch` and `useSelector` hooks in a *container* component.
2. You can refactor your project to use the `useDispatch` and `useSelector` hooks within the component itself.

You can choose either method to refactor your project. At this point, you are a full-fledged React developer - it's time for you to start planning your own

React code! Talk things through with your partner and decide which way you would like to begin refactoring your project. Remember, as you are using this project as practice for implementing Redux hooks, you can always implement Redux hooks into a container component for one component, and then implement Redux hooks directly in another component for the next component. Just make sure to choose one method to stick with for your personal React projects!

Version 1: Using Redux hooks in a container component

If you chose to use the `useDispatch` and `useSelector` hooks in a *container* component, this means you should use a namespace to import actions into each component file. For example, in the `LogoutButton.js` file, you currently have the following import statement that imports the `logout` action creator function:

```
import { logout } from '../actions/authentication';
```

In order to remove confusion about whether an invocation of `logout()` is invoking the `logout` prop or the `logout` action creator function, you can update the import statement for your `logout` action to use an `AuthAction` namespace:

```
import * as AuthAction from '../actions/authentication';
```

This way, you can create a container component to replace what is happening under the hood with the `mapStateToProps`, `mapDispatchToProps`, and `connect` functions. You can reference the `logout` action with the `AuthAction` namespace, like so: `AuthAction.logout`.

Based on the `mapStateToProps` and `mapDispatchToProps` functions in the `LogoutButton.js` file, you can tell that the component is accessing Redux by receiving `loggedOut` and `logout` props:

```
const mapStateToProps = state => {
  return {
    loggedOut: !state.authentication.token,
  };
};

const mapDispatchToProps = dispatch => {
  return {
    logout: () => dispatch(logout()),
  };
};

export default connect(mapStateToProps, mapDispatchToProps)(LogoutButton);
```

Take a moment to import the `useSelector` and `useDispatch` from the Redux library into the file:

```
import { useDispatch, useSelector } from 'react-redux';
```

Now you'll write a container component that will replace the `mapStateToProps`, `mapDispatchToProps`, and `connect` functions! Start by setting up the container component that returns the `LogoutButton` component to use the `useDispatch` prop. You'll also want to have the `LogoutButton.js` file export the `LogoutButtonContainer` component (instead of the higher-order component returned by the `connect` function):

```
const LogoutButtonContainer = () => {
  const dispatch = useDispatch();

  return <LogoutButton />;
};

export default LogoutButtonContainer;
```

Feel free to visit the [Redux Hooks documentation](#) to view `useDispatch` [examples](#).

Now that you have the container component and `dispatch` set up, you can pass dispatched version of the `logout` action as a prop into the `LogoutButton` component:

```
const LogoutButtonContainer = () => {
  const dispatch = useDispatch();
  const logout = () => dispatch(AuthAction.logout());

  return <LogoutButton logout={logout} />;
};
```

At this point, the container component is taking care of what the `mapDispatchToProps` function took care of! Now let's use the `useSelector` hook to take care of what the `mapStateToProps` function took care of:

```
const LogoutButtonContainer = () => {
  const dispatch = useDispatch();
  const logout = () => dispatch(AuthAction.logout());
  const loggedIn = useSelector(state => !state.authentication.token);

  return <LogoutButton logout={logout} />;
};
```

Feel free to visit the [Redux Hooks documentation](#) to view `useSelector` [examples](#).

Now that you've gone over how to create a container component that implements Redux Hooks for the `LogoutButton` component, follow the same pattern to implement Redux hooks into container components for your `LoginPanel`, `PokemonDetail`, `PokemonForm`, and `PokemonBrowser` components. Feel free to practice implementing Redux hooks directly within a component instead!

Version 2: Using Redux hooks from within a component

If you chose to use the `useDispatch` and `useSelector` hooks within the component itself, you'll need to do some refactoring so that your component doesn't receive any props. Instead of receiving slices of state and dispatchable action functions as props, you will use the `useSelector` hook to access a slice of state from within the component and the `useDispatch` hook to dispatch actions from within the component.

Based on the `mapStateToProps` and `mapDispatchToProps` functions in the `LogoutButton.js` file, you can tell that the component is accessing Redux by receiving `loggedIn` and `logout` props.

```
const mapStateToProps = state => {
  return {
    loggedIn: !state.authentication.token,
  };
};

const mapDispatchToProps = dispatch => {
  return {
    logout: () => dispatch(logout()),
  };
};

export default connect(mapStateToProps, mapDispatchToProps)(LogoutButton);
```

Take a moment to import the `useSelector` and `useDispatch` from the Redux library into the file.

```
import { useDispatch, useSelector } from 'react-redux';
```

Now you'll want to remove all the props that the `LogoutButton` receives. Instead of receiving props to access the `loggedIn` state and dispatched

`logout` function, you'll use the `useSelector` and `useDispatch` hook you just imported into the file. At this point, your `LogoutButton` component should look something like this:

```
const LogoutButton = () => {
  if (loggedOut) {
    return <Redirect to="/login" />;
  }

  return (
    <div id="logout-button-holder">
      <button onClick={handleClick}>Logout</button>
    </div>
  );
};
```

Now you'll use Redux hooks within the `LoginButton` component so that you can remove the `mapStateToProps`, `mapDispatchToProps`, and `connect` functions! Instead of receiving a `loggedOut` prop, you'll use the `useSelector` hook to access the state's `authentication.token`.

```
const LogoutButton = () => {
  const loggedOut = useSelector(state => !state.authentication.token);

  // CODE SHORTENED FOR BREVITY
};
```

Feel free to visit the Redux Hooks documentation to view `useSelector` [examples](#).

Notice how the `logout` thunk action creator function has already been imported into your `LogoutButton.js` file. You'll use `useDispatch` hook to return a reference to the `dispatch` function from the Redux store:

```
const dispatch = useDispatch();
```

Then you can use the `dispatch` function to dispatch the `logout` function:

```
const LogoutButton = () => {
  const loggedOut = useSelector(state => !state.authentication.token);
  const dispatch = useDispatch();
  const handleClick = () => dispatch(logout());

  // CODE SHORTENED FOR BREVITY
};
```

Feel free to visit the Redux Hooks documentation to view `useDispatch` [examples](#).

Lastly, you'll want to remove the `mapStateToProps` and `mapDispatchToProps` functions from the file, and replace the `connect` function to an export statement that exports the `LoginButton` component by default:

```
export default LoginButton;
```

Now that you've gone over how to refactor your `LogoutButton` component, follow the same pattern to implement Redux hooks into your `LoginPanel`, `PokemonDetail`, `PokemonForm`, and `PokemonBrowser` components. Feel free to practice creating a container component that utilizes Redux hooks instead!

Router hooks: `useParams`

Notice the references to the React Router `match` prop accessed in your `PokemonBrowser` and `PokemonDetail` components. Instead of having your component take in a `match` prop to access the route parameters, you'll implement the `useParams` prop and use object destructuring to access the `pokemonId` parameter in the `PokemonBrowser` component and the `id` parameter in the `PokemonDetail` component. Feel free to visit React Router documentation to view examples of using the `useParams` [hook](#).

Once you have finished refactoring, take a moment to commit your changes to your `redux-hooks-app` branch:

```
git add .
git commit -m "Refactor app to implement redux hooks"
```

Now that you have practiced refactoring your application to implement Redux hooks, it's time to work on a Context-based project utilizing React's `useContext` hook! In the next phase, you'll branch off from your application's main branch to create a new `context-hooks-app` branch for the project.

Pokedex Hooks Project: Phase 3

At this point, you've managed a state-based project with the `useState` and `useEffect` hooks and a Redux-based project with the `useSelector` and `useDispatch` hooks. Now you'll work off of the state-based project you built in Phase 2 to manage your application's state with the `useContext` hook!

Begin by creating a new branch for your Context-based application off of your main branch:

```
git checkout master
git checkout -b context-hooks-app
```

Using the `useContext` hook to manage application state

Think of how you created a Redux cycle to pass Pokedex information through your components. Now you'll manage your application's global information by using

React Context instead! Remember that you still generate context with the `createContext` function, just as you would for class components. You also still use `<Context.Provider>` components to set the `value` of your context object.

Think of how you would make use of the `useContext` hook instead of Redux's `connect()` function to pass slices of state as well as functions to update the global state. As you use the `useEffect` hook for side effect operations, think of conditions where you would want to prevent the effect from running. Remember to set the variables that determine these conditions in the `useEffect` hook's dependency array.

Providing context

Begin by creating a `PokemonContext` with the `createContext` function from React. Then you'll need to set the context value with a `<Context.Provider>` component by making a wrapper Provider component for the `App` component. Create an `AppWithContext` component as the wrapper component for `App`. The wrapper component will have the following slices of state:

- `pokemon` - defaults to an empty array.
- `singlePokemon` - defaults to `null`.
- `authToken` - defaults to the `state-pokedex-token` item stored in `localStorage`.
- `needLogin` - defaults to the *truthyness* of the `state-pokedex-token` item stored in `localStorage` (hint: you can use the `double not !!` notation).

The wrapper component will also pass the following functions as the context `value`:

- `login(token)` - to set the `state-pokedex-token` item in `localStorage`, update the `authToken` state, and update the `needLogin` state to `false`.
- `loadPokemon()` - to fetch all pokemon and update the `pokemon` state.
- `getOnePokemon(id)` - to fetch one pokemon and update the `singlePokemon` state.

After you have set up the wrapper component, make sure to replace the `App` that is rendered in your `index.js` file with your new `AppWithContext` wrapper component. Now it's time to change your application from being a state-based application to a context-based application.

Begin by removing all props that are passed between components. You'll use the hooks and the `PokemonContext` value to manage the global state of your application instead. Make sure to even remove the `match` prop you access in the `PokemonDetail` component. You'll use the `useParams` hook from [React Router v5.1](#) instead of the `match` prop.

Consuming context with the `useContext` hook

Your application's consuming components should access the `PokemonContext` through using the `useContext` hook. Feel free to reference the [Hooks API Reference](#) to revisit the documentation on the `useContext` hook. As a reminder, the `useContext` hook replaces the `static contextType` property of class components:

`static contextType`

```
// Receive access to context in class components:
static contextType = PokemonContext;

// Access context with the `contextType` property:
this.context
```

`useContext`

```
// Receive access to context with React Hooks in function components:
const context = useContext(PokemonContext);
```

```
// Access context with the `useContext` hook:
context
```

Now it's time to set up how your application components *consume* the `PokemonContext` !

Feel free to console log the `context` in any component you are accessing the `PokemonContext` . It could be helpful to create an application state logging system like so:

```
const context = useContext(PokemonContext);
console.log(context);
```

This way upon the mounting of a component, you have a general sense of the `context` object the component is receiving. Since you'll be using Hooks and Context to manage your user authentication, you may need to clear your `localStorage` items to reset your application to allow for future testing and debugging. As a reminder, you can go to the `Application` tab of your developer tools to find a `Storage` section with your `Local Storage` items. There you can right click to delete all items stored in `localStorage` .

App

Take a moment to compare the code that currently lives in your `AppWithContext` with the code that lives in you `App` component. Notice how there is a lot of duplicated logic. This is because you `App` was the main component managing your application's state-based information. Now that you have moved all the logic to your `AppWithContext` component, you can refactor your `App` component to simply use the `useContext` hook to pass the `needLogin` value to the `<PrivateRoute>` it renders. You can also remove all other props passed through the routes. Your refactored `App` component should look something like this:

```
// App.js
import React, { useContext } from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom';

import { PrivateRoute } from './routesUtil';
import LoginPanel from './LoginPanel';
import PokemonBrowser from './PokemonBrowser';
import { PokemonContext } from './PokemonContext';

const App = () => {
  const { needLogin } = useContext(PokemonContext);

  return (
    <BrowserRouter>
      <Switch>
        <Route path="/login" component={LoginPanel} />
        <PrivateRoute
          path="/"
          component={PokemonBrowser}
          needLogin={needLogin}
        />
      </Switch>
    </BrowserRouter>
  );
}

export default App;
```

LoginPanel

Let's begin by refactoring your `LoginPanel` component! The `LoginPanel` should access the context's `login` function. Feel free to also access the `authToken` value for testing purposes. Note that you can also destructure the context object, like so:

```
const { login, authToken } = useContext(PokemonContext);
console.log(authToken);
```

You'll want to make sure that your `LoginPanel` has the following slices of state:

- `loggedIn` - defaults to `false`.
- `email` - defaults to `'demo@example.com'`.
- `password` - defaults to `'password'`.

Your `LoginPanel` component should also hold the following three functions:

- `handleSubmit` - to make a fetch request to your API, update the `loggedIn` state, and invoke the context's `login` function with the `token` from the fetch response.
- `updateEmail` - to update the `email` slice of state.
- `updatePassword` - to update the `password` slice of state.

Lastly, your `LoginPanel` should redirect logged in users to the home page, based on the `loggedIn` slice of state:

```
if (loggedIn) return <Redirect to="/" />;
```

Now you'll want to set up your `PokemonBrowser` component before testing the login flow and home page redirection - you'll hit a lot of errors if you don't refactor your `PokemonBrowser` correctly first.

PokemonBrowser

At this point, your `PokemonBrowser` component should look something like this:

```
import React from "react";
import { NavLink, Route } from "react-router-dom";
import { imageUrl } from "../config";
import PokemonDetail from "../PokemonDetail";

const PokemonBrowser = ({ pokemon, token }) => {
  if (!pokemon) return null;

  return (
    <main>
      <nav>
        {pokemon.map((poke) => {
          return (
            <NavLink key={poke.name} to={`/${pokemon}/${poke.id}`}>
              <div className="nav-entry">
                <div className="nav-entry-image"
                  style={{
                    backgroundImage: `url('${imageUrl}${poke.imageUrl}');`
                  }} />
                <h1>{poke.name}</h1>
              </div>
            </NavLink>
          );
        })}
      </nav>
      <Route
        path="/pokemon/:id"
        render={(props) => <PokemonDetail {...props} token={token} />
      />
    </main>
  );
};

export default PokemonBrowser;
```

You won't need to refactor any existing code within component, except removing the props it receives. You'll simply use the `useContext` hook to access the `PokemonContext` and use the `useEffect` hook to update the context's `pokemon`.

Begin by having your `PokemonBrowser` component access the context's `pokemon` and `loadPokemon` function:

```
const { pokemon, loadPokemon } = useContext(PokemonContext);
console.log(pokemon);
```

You'll want to update the global state by invoking `loadPokemon` upon load. Since data fetch is considered a *side effect operation*, you'll invoke the `loadPokemon` function within a `useEffect` hook:

```
useEffect(() => {
  loadPokemon();
}, []);
```

Note that the hook's *dependency array* is empty. If you start your server, you'll notice that your application will be stuck in an infinite loop to fetch pokemon! Think of what conditions you want your `loadPokemon` function to be invoked (hint: think of how to use the length of the `pokemon` array).

Take a moment to test the user login flow. You want to be redirected to view the `PokemonBrowser` component. You also want to keep an eye on your backend database logs. Make sure that you are setting correct variables to optimize the fetch calls made from the `useEffect` hook from your `PokemonBrowser` !

PokemonDetail

Instead of using the `match.params.id` prop, your `PokemonDetail` component will make use of the `useParams` hook from [React Router v5.1](#)! Begin by using the `useContext` hook to give the component access to the context's `singlePokemon` and `getOnePokemon` function.

Note that you can also rename the object keys to prevent the need to refactor your rendered JSX. In the snippet below, the context's `singlePokemon` is renamed to be `pokemon`.

```
const { singlePokemon: pokemon, getOnePokemon } = useContext(PokemonContext);
console.log(pokemon);
```

This way, you won't need to refactor any of the render code to render the `pokemon` in the `PokemonContext` !

Have your `PokemonDetail` component update the global state by invoking the `getOnePokemon` function with the `id` from the route parameters upon load. Import the `useParams` hook from the `react-router-dom` package:

```
import { useParams } from 'react-router-dom';
```

Now you can simply invoke the function and destructure the `params` object it receives!

```
const { id } = useParams();
```

You'll need to use a `useEffect` hook to fetch a pokemon based on the `id` from the route parameters. Just like in your `PokemonBrowser` component, you need to determine what variables to place in the dependency array so that your application is not stuck in an infinite fetch loop!

In your `useEffect` hook, you'll need to check two conditions. If the single `pokemon` is *falsey*, have your component invoke the `getOnePokemon` function with the `id` parameter. If the pokemon's ID is not equal to the route parameter `id`, also have your component invoke the `getOnePokemon` function to fetch a specific pokemon! Remember that the `id` from your route parameters is currently a string, so you'll need to parse the `id` in order to make a valid comparison to the pokemon's ID.

After you have finished refactoring your state-based React application built with class components to a context-based React application built with function

components and React Hooks, compare your context-based application to the redux-based solution. Using Redux instead of Context results in a lot of boilerplate code in your application. React 16 revamped the Context API and deemed the `useContext` hook as a basic hook to improve React's built-in state management.

Redux is a large library with a lot of conceptual knowledge involved. In the next project, you will dive deeper and have more practice with implementing the Redux library for state management. Although you are free to use either Context or Redux to manage the application state of your project next week, you can take today's bonus project as a chance to dive in deeper and truly learn Redux. Understanding all the conceptual knowledge behind the library will help you architect your own React project and plan your application's state management.

WEEK-15 DAY-5

WebSockets and Project Planning

WebSockets Learning Objectives

WebSockets enable two-way communication between the user's browser (the client) and a server. They can be used to enable dynamic, interactive web experiences. After reading and practicing, you should be able to:

- Use the WebSockets API to create a new WebSocket connection to a server

- Create a WebSocket `onopen` event handler function to detect when the connection has been opened
 - Create a WebSocket `onmessage` event handler function to detect and process messages sent by the server
 - Create a WebSocket `onerror` event handler function to detect when an error has occurred
 - Use the WebSocket `send()` method to send messages to the server
 - Recall that WebSocket message data can be sent as JSON formatted string
 - Recall that WebSocket messages usually have a "type" associated with them so the client can determine how to process them
 - Use the WebSocket `close()` method to close the connection to the server
 - Create a WebSocket `onclose` event handler function to detect when the connection to the server has been closed
 - Use the `ws` package to create a standalone WebSocket server
 - Use the `ws` package to create a WebSocket server that shares a Node.js `http` server with an Express application
 - Create an `connection` event handler listener method to detect when a client has connected to the WebSocket server
 - Create an `close` event handler listener method to detect when a client has closed the connection to the WebSocket server
 - Use the WebSocket `send()` method to send a message to a client
-

WebSockets Overview

Up until now, you've seen communication between the Web browser and your backend server occur in the *request/response cycle* of HTTP 1.1.

 HTTP Exchange

The *Client* makes an HTTP request, like **GET /home HTTP/1.1**. The *Server* receives that request, translates it, and returns an HTTP response, like **HTTP/1.1 200 OK**. One request, one response. That is great for getting data and asking the server to create new resources, but it does not support the demands of Web applications that need "real-time communication" or to receive messages from the server *without* an HTTP request. The [WebSockets](#) standard fills that hole. (That's a link to the RFC. It's ... dense.)

Check out this [link to caniuse.com](#) that tracks the support of WebSockets (and a whole bunch of other things) in browsers for the desktop and mobile. You can see that *everything* supports WebSockets (except Opera Mini which fails to support pretty much anything, stupid Opera Mini).

Since the technology is now well-supported, it makes sense to learn it so that you can do *amazing* things in your Web application.

In this article, you will learn about how WebSockets work from the perspectives of the communication between the client and the server.

Just one thing: there are *lots* of libraries out there for you to use that make this WebSocket thing "easy". In the same way that this curriculum challenges you to use `fetch` rather than some other AJAX library like `axios`, learning WebSockets teaches you about the *technology* and how it works. Once you know that, then you can use *any* library (like [socket.io](#)) to ease your development. But, giving you the deeper knowledge is what this is providing for you.

Key points about Web sockets

Persistent Connections

In the traditional model of request/response, the client makes a connection to the server, makes the request, the server responds on the same connection, then

the connection can be closed. The next time your browser wants to make a request to the Web server, it may need to establish that connection, again.

WebSockets create a *persistent connection*, one that doesn't close unless it doesn't get used. This means that the TCP/IP handshake that needs to occur between the browser and server does not need to happen with every single request. This has two benefits:

1. Your front-end application does not need to establish that connection *every single time*. This makes the application feel snappier.
2. The connection is *two-way*! While it's open, the server can send messages to the client *whenever it wants to*! That is mind-blowing because HTTP 1.1 cannot support that without jumping through crazy hoops that can cause your server to crash.

The way it happens is an extra HTTP header in the HTTP request. Here's an example.

```
GET ws://WebSocket.example.com/ HTTP/1.1
Origin: http://example.com
Connection: Upgrade
Host: WebSocket.example.com
Upgrade: WebSocket
```

So, checkout two things about this:

1. The protocol is "ws" which, you can probably guess, means "WebSocket". (There's also a "wss" which is like "HTTPS" but for WebSockets Secure.)
2. The "Connection" header tells the server that the browser is requesting an upgrade for the normal HTTP 1.1 connection
3. The "Upgrade" header tells the server that the browser specifically wants a WebSocket

If the server supports WebSockets, it says "COOL!" and returns something like the following headers in the response.

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Thu, 7 May 2020 17:07:34 GMT
Connection: Upgrade
Upgrade: WebSocket
```

This confirms that the server is good with upgrading the connection. When both the client and server agree, they just don't close the connection.

Boom! Persistent!

Messages, not requests and responses

Once the connection exists between the browser and the server, either of the two can send a *message* over the connection. It's a *message* with a sender and a receiver. It is *not* a request/response. There is no request. There is no response. There are just two actors sending messages back and forth, like two kids in school passing notes back and forth in class. The server doesn't have to wait for a request to send a message. The client can send a message *without* the expectation of a response.

Just like in TCP/IP, when data gets chopped up into packets and datagrams, messages over WebSockets get chopped up into *frames*. Each frame contains extra information to help ensure the integrity of the message as it traverses between sender and recipient. It's not super important to know what those parts are because you're not writing code to implement the standard; instead, the browser will do it for you automatically, just like using `fetch` means you don't have to format the HTTP request.

Client-side code

Just like the browser has the `fetch` method to easily make HTTP 1.1 requests, it provides a `WebSocket` class for you to create objects that manage the

connection between the browser and the server. You just give the constructor the WebSocket URL that you want your browser to connect to.

```
// This is EXAMPLE CODE ONLY!  
// There is no sockets.example.com!  
const socket = new WebSocket('wss://sockets.example.com');
```

Now, with `fetch`, that *sends* a request and, when a response comes back, the `Promise` gets fulfilled and you do stuff with it. That's not how `WebSocket` objects work. They *can't* work that way.

Instead of a `Promise`, you add event listeners to the `WebSocket` object in the same way that you add event listeners to `input` or `button` elements to capture specific kinds of events. For the `WebSocket`, the events are

- **message** fired when a message fully arrived over the `WebSocket` from the server
- **close** fired when the `WebSocket` closes for some reason, the status code being in the "code" property of the event and the reason in the "reason" property of the event object
- **error** fired when the `WebSocket` can't even connect
- **open** fired when the `WebSocket` opens

Then, the `WebSocket` object has two methods for you to use, `send` to send a message to the server, and `close` to close the connection. Here's what some code could look like that uses that `socket` opened above.

```
// This is EXAMPLE CODE ONLY!  
  
// When the socket is open, send a message!  
socket.addEventListener('open', () => socket.send('I am LEGENDARY!'));  
  
// When you get a message, add it to your state store.  
socket.addEventListener('message', event => {  
  dispatch(gotMessage(event.data));  
});  
  
// Print out that something bad happened  
socket.addEventListener('error', () => {  
  console.error('Something bad happened... :-(');  
});  
  
// When the socket closes, update the state  
// of the application!  
socket.addEventListener('close', () => {  
  dispatch(justDisconnected());  
});
```

Note: just like with DOM elements where you could use `el.onclick = () => {...}` to add an event handler. You can do something like `socket.onmessage = () => {...}`, too. But, that's just not nice because you can't add more than one listener. So, if you see that in the documentation, somewhere, remember that you can always use `addEventListener` rather than the `on«event»` properties.

All of that is just provided for you in the browser! There's a lot of code under all of that to allow your JavaScript that easy-to-use API! Thanks, browser makers!

You can give it a shot yourself. Create a new HTML 5 file with all of the normal stuff and add this code in there.

In the body of the document, put this.

```

<div>
  <button id="connect">Connect</button>
  <button id="send-message">Send</button>
  <button id="disconnect">Disconnect</button>
</div>
<div id="messages"></div>

```

Now, create a `script` element *after* the content you just added (so you don't have to wait for "DOMContentLoaded"). This is just regular-old DOM code with the socket message stuff in there, too. Have a look and try it out! Change the code so that you can see how changes affect it!

This code uses a *real* WebSocket server, `wss://echo.websocket.org`!

```

const connect = document.getElementById('connect');
const disconnect = document.getElementById('disconnect');
const sendMessage = document.getElementById('send-message');
const messages = document.getElementById('messages');

let socket = null;

connect.addEventListener('click', () => {
  messages.innerHTML += `<p>Opening WebSocket...</p>`;
  socket = new WebSocket("wss://echo.websocket.org/");

  socket.addEventListener('open', () => {
    messages.innerHTML += `<p>CONNECTED!</p>`;
  });

  socket.addEventListener('message', event => {
    messages.innerHTML += `<p>Received "${event.data}"</p>`;
  });

  socket.addEventListener('error', () => {
    messages.innerHTML += `<p>ERROR</p>`;
  });

  socket.addEventListener('close', () => {
    messages.innerHTML += `<p>Socket closed</p>`;
    socket = null;
  });
});

disconnect.addEventListener('click', () => {
  if (!socket) {
    messages.innerHTML += `<p>Socket not open.</p>`;
    return;
  }

  socket.close();
});

sendMessage.addEventListener('click', () => {
  if (!socket) {
    messages.innerHTML += `<p>Socket not open.</p>`;
    return;
  }

```



```
}
messages.innerHTML += `<p>Sending "WebSockets are cool!"</p>`;
socket.send('WebSockets are cool!');
});
```

Here's an interesting thing. After you play around with the code, refresh the page and connect to the server. Then, just wait. Likely, eventually, the socket will close due to disuse. Many libraries (like socket.io) keep the connection "warm" by sending little *ping* methods to the server to let the server know that it really does want to keep that connection open. If it doesn't close, then you have a *really* good and stable Internet connection!

Server-side code

If that's the client-side code above, the question that might be bothering you is "How hard is the server-side code?" Well, luckily, it's just about the same level of ease with the [ws](https://www.npmjs.com/package/ws) package for Node.js.

Because WebSockets are a browser-based technology, the implementations that you will find on the server can vary widely. Luckily, the **ws** API is an event-driven API, too. It provides these events for you to use to build a WebSocket server using the `Server` object.

- The **connection** event occurs when the HTTP handshake completes but the connection has *not* upgraded. It's callback receives a `WebSocket` that allows you to communicate with the client.
- The **headers** event which allows you to inspect and modify headers before they are sent back to the client.
- The **listening** event which fires when the underlying server is bound to a *network socket*, not a Web socket.

Then, the server has a `close` method which lets it shut down. It's got some other methods, too, about handling upgrades and stuff, which are outside the

scope of this article. You are encouraged to go check out the API docs in a later article.

Those last two are some pretty low-level events that you won't necessarily have to pay attention to unless you're doing some *really* advanced stuff. However, you *will* want to pay attention to the **connection** property because that is how you know a client is connected. Then, when the connection gets upgraded, the callback gets a `WebSocket` object so that your server can send messages to the browser.

Here's the code to write an "echo" server like you just used in the client-side stuff above. Put this in a file, install "ws" using `npm install ws`, and run it with `node «filename»`. Then, change the URL in the HTML file that you created from `wss://echo.websocket.org/` to `ws://localhost:8080`. Everything still works!

```
const WebSocket = require('ws');

const server = new WebSocket.Server({ port: 8080 });

server.on('connection', websocket => {
  console.log('client connecting..');

  let interval = null;
  setInterval(() => websocket.send('Hello?'), 1000);

  websocket.on('message', message => {
    console.log('received: %s', message);
    websocket.send(message);
  });

  websocket.on('close', () => {
    console.log('Connection closed.');
    clearInterval(interval)
  });
});
```

You can see that a server gets created using port 8080. That server then waits for connections with `server.on('connection', ...)`. When the connection occurs, the callback gets called and `websocket` gets set to the actual `WebSocket` instance that you can use to send (and receive) messages to (and from) the client. Then, it creates an interval that sends a "Hello?" message to the client every second or so.

You subscribe to messages using `websocket.on('message', ...)`. When a message arrives from the browser, the callback gets called with the content of the data in the `message` variable. Normally, that'll be a JSON-formatted string that you can use to do things with your code.

Finally, when the `WebSocket` object closes, it prints a message to the console and clears that interval.

That's how nice **ws** makes it to write WebSocket-enabled. Thank you, **ws**!

A really cool thing about **ws** is that it can track clients for you when you create the `Server` object by passing in the `clientTracking` option when you construct it. Then, the `clients` property on the `Server` object will have all of the clients on it so you can broadcast messages to *everyone*!

What you've learned

1. WebSockets are a persistent connection between the browser and server
2. It's a two-way connection, messages can flow in both directions
3. The "Connection" header is used to request an upgrade. The "Upgrade" header specifically requests a WebSocket.
4. Client-side code is an event-driven model with **open**, **close**, **message**, and **error** events
5. Server-side code is an event-driven model that waits for **connection** events on a `Server` object which, then, provides a `WebSocket` object nearly identical to the `WebSocket` used on the client side.

WebSocket Server Applications

Now that you know how to use WebSockets on the client, it's time to learn how to use WebSockets on the server. To enable support for WebSockets on the server, you'll use the `ws` [npm package](#), a WebSocket server implementation for use with Node.js applications.

The `ws` package provides both a *server* and a *client* implementation. The client implementation is only intended for use on Node.js, to enable *server* to *server* WebSocket connections. When reading [the documentation for the `ws` package](#), focus on the following *server* examples:

- [Simple Server](#)
- [Server Broadcast](#)
- [Example Application: Server Stats](#)
- [Example Application: Express Session Parse](#)

After reading [the documentation for the `ws` package](#), you should be able to:

- Use the `ws` package to create a standalone WebSocket server;
 - Use the `ws` package to create a WebSocket server that shares a Node.js `http` server with an Express application;
 - Create an `connection` event handler listener method to detect when a client has connected to the WebSocket server;
 - Create an `close` event handler listener method to detect when a client has closed the connection to the WebSocket server; and
 - Use the WebSocket `send()` method to send a message to a client.
-

WebSocket Client Applications

WebSockets enable two-way communication between the user's browser (the client) and a server. Normally, the server only responds to client requests. When using WebSockets, once the client has opened a connection with the server, the server can send messages to the client.

A good place to start learning about Websockets on the Web is with the [Writing WebSocket client applications](#) article on MDN web docs.

After reading this article, you should be able to:

- Use the WebSockets API to create a new WebSocket connection to a server;
 - Create a WebSocket `onopen` event handler function to detect when the connection has been opened;
 - Create a WebSocket `onmessage` event handler function to detect and process messages sent by the server;
 - Create a WebSocket `onerror` event handler function to detect when an error has occurred;
 - Use the WebSocket `send()` method to send messages to the server;
 - Recall that WebSocket message data can be sent as JSON formatted string;
 - Recall that WebSocket messages usually have a "type" associated with them so the client can determine how to process them;
 - Use the WebSocket `close()` method to close the connection to the server; and
 - Create a WebSocket `onclose` event handler function to detect when the connection to the server has been closed.
-

Tic-Tac-Toe Online Project

In this project, you'll get a chance to practice your newly acquired WebSockets skills by building an interactive game! Earlier, you built a Tic-Tac-Toe game that could be played locally, within a single browser session. Now you'll extend the game to use WebSockets so that it can be played online, across two different browser sessions. You'll also convert the client-side code to use React components.

Phase 0: Reviewing the application design and architecture

Here's a high level description of the design and architecture of the application:

- Node.js and Express will be used to create a simple web application.
 - The server will serve static files from a `public` directory. In production the `public` directory would contain a production build of the React client application.
- The `ws` [npm package](#) will be used to create a WebSocket server.
 - The WebSocket server will enable two-way communication between two users as they play a game of tic-tac-toe.
 - All game state will live on the server. As players take turns, the game state will be updated on the server and the WebSocket server will be used to broadcast state updates to the clients to keep their game boards in sync.
- The client application will be built using React web components.
 - Some components will require state, but you're welcome to use either class or function components with Hooks.
 - Global state requirements in the client application will be relatively simple, so while you could use a Context or Redux to maintain your application's state, neither one is really a requirement to use.
- Players will be prompted for their "player name" when they first browse to the web application.

- To keep things simple, the server will start a new game as soon as two players have connected to the server.
- If a player unexpectedly quits the game (i.e. closes their browser) then the game will abort for the remaining player.

The initial version of this project will be fun and challenging, giving you ample opportunity to practice your WebSocket skills. There's also lots of interesting ways that this project can be extended beyond the initial requirements.

Ready to get started building your online tic-tac-toe game? **Let's go!**

Phase 1: Setting up the client and server projects

This project will actually be split into two projects: a Node.js project for the Express server application and a Create React App project for the client application.

Stubbing out the server project

To save you a bit of time, we've provided you with a repo of starter files for the server project. Create a top level folder for your project; name it something like `tic-tac-toe-online`. Browse into that folder and clone this repository:

<https://github.com/appacademy-starters/tic-tac-toe-online-starter.git>

Once the repo has finished cloning, you can browse into the `server` folder and install the server project's dependencies by running the command `npm install`. Once the dependencies have finished installing, test the application by running `npm start` and browsing to `http://localhost:8080/`. You should see a very simple web page displaying the heading "Tic-Tac-Toe Online".

Take a moment to review the `app.js` file's contents:

```
// ./app.js

const express = require('express');
const path = require('path');
const { createServer } = require('http');
const morgan = require('morgan');

const { port } = require('./config');

const app = express();

app.use(morgan('dev'));
app.use(express.static(path.join(__dirname, 'public')));

app.get('*', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

const server = createServer(app);

server.listen(port, () => console.log(`Listening on http://localhost:${port}`));
```

This simple Express application will:

- Attempt to match incoming requests to static files located in the `public` folder; and
- If a matching static file isn't found, then the `./public/index.html` file will be served for all other requests.

In the next section, you'll use Create React App to create a client project. Once you've built your React client application, you can create and copy the production build into the `public` folder. This allows the one Express application to serve both the client and server parts of the application.

Remember that configuring Express to serve the `./public/index.html` file for any request that doesn't match a static file, allows you to "deep link" to any of your React application's routes (if you use routing in your client application).

One thing to note about the above `app` module, is that the `http.createServer` method is being used to create the HTTP server instead of calling the `listen` method on the Express Application (`app`) object. In a bit, you'll see how using this approach will allow you to use the same server for both HTTP and WebSocket requests.

Stubbing out the client project

After stubbing out the server project, browse back up to the top level project folder and use Create React App to create your client project:

```
npx create-react-app client --template @appacademy/simple
```

When the command completes, browse into the `client` and run `npm start`. The Create React App development server should start and open your client application into your default browser. If it doesn't automatically happen, you can manually open a browser and browse to `http://localhost:3000/`. When the page loads, you should see a heading displaying the text "Hello world!"

Phase 2: Stubbing out the React components

The React client application will be relatively simple: it'll contain just three components (at least initially):

- An `App` component (this is already in your project);
- A `Home` component; and
- A `Game` component.

Go ahead and stub out the `Home` and `Game` components within a `components` folder. The `Home` component will require state so use a class component or a function component with the `useState` Hook (we'll be using Hooks in the instructions). The `Game` component doesn't require any state, so a function component will work fine.

```
// ./src/components/Home.js

import React, { useState } from 'react';

const Home = () => {
  return (
    <h2>Home</h2>
  );
}

export default Home;
```

```
// ./src/components/Game.js

import React from 'react';

const Game = () => {
  return (
    <h2>Game</h2>
  );
}

export default Game;
```

Import the `useState` Hook into the `App` module and refactor the `App` component from a function declaration to an arrow function expression. Then update the `App` component to import and render the `Home` and `Game` components:

```
// ./src/App.js

import React, { useState } from 'react';

import Home from './components/Home';
import Game from './components/Game';

const App = () => {
  return (
    <div>
      <h1>Tic-Tac-Toe Online</h1>
      <Home />
      <Game />
    </div>
  );
}

export default App;
```

Also notice that application heading was updated to "Tic-Tac-Toe Online". Do the same for the title in the `./public/index.html` file:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Tic-Tac-Toe Online</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

Go ahead and run your client application again (`npm start`) to ensure that the `Home` and `Game` components render as expected. Don't worry that they're both displaying at the same time; we'll fix that in a bit.

Phase 3: Rendering the game board

Before turning our attention to using WebSockets to implement the interaction between the client and server, let's update the `Game` component to render the game board.

Adding the styles

To start, add a CSS module file named `Game.module.css` to the `./src/components` folder containing the following:

```

/* ./src/components/Game.module.css */

.game {
  margin: auto;
  width: 402px;
}

.players {
  display: grid;
  height: 20px;
  width: 400px;
  grid-template-columns: 1fr 1fr;
  grid-template-rows: 1fr;
}

.tic_tac_toe_board {
  display: grid;
  height: 400px;
  width: 400px;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 1fr 1fr 1fr;
  background-color: black;
  margin: 32px 0;
}

.actions {
  display: flex;
}

.announcement {
  font-size: 1.4em;
  text-align: center;
}

.col_1 {
  justify-self: start;
}

.col_2 {
  justify-self: center;
}

```

```

.col_3 {
  justify-self: end;
}

.row_1 {
  align-self: start;
}

.row_2 {
  align-self: center;
}

.row_3 {
  align-self: end;
}

.spacer {
  flex: 1 0 0px;
}

.square {
  background-color: white;
  height: 130px;
  width: 130px;
}

```

Then add the following global styles to the `index.css` file:

```

/* ./src/index.css */

body, button, input {
  font-family: Arial, Helvetica, sans-serif;
}

button, input {
  font-size: 1.1em;
}

```

The initial game board

In the `Game` component, be sure to import your CSS module file, then update the `render` method to this:

```
// ./src/components/Game.js

import React from 'react';
import styles from './Game.module.css';

const Game = () => {
  return (
    <div className={styles.game}>
      <div className={styles.players}>
        <div>Player X: { /* TODO Render player 1 name */}</div>
        <div>Player O: { /* TODO Render player 2 name */}</div>
      </div>
      <h3 className={styles.announcement}>TODO</h3>
      <div className={styles.tic_tac_toe_board}>
        { /* TODO Render game board squares */}
      </div>
    </div>
  );
}

export default Game;
```

Later in the project, once we have the game state available to us, we'll replace the `TODO`s with the player names. We'll also be able to render some action buttons below the game board when a game is ended to allow users to play again or quit.

Rendering the game board squares

For now, let's turn our attention to rendering the game board squares. In the original tic-tac-toe project, the game board squares were represented by the following HTML:

```
<div id='square-0' class='square row-1 col-1'></div>
<div id='square-1' class='square row-1 col-2'></div>
<div id='square-2' class='square row-1 col-3'></div>
<div id='square-3' class='square row-2 col-1'></div>
<div id='square-4' class='square row-2 col-2'></div>
<div id='square-5' class='square row-2 col-3'></div>
<div id='square-6' class='square row-3 col-1'></div>
<div id='square-7' class='square row-3 col-2'></div>
<div id='square-8' class='square row-3 col-3'></div>
```

In the original tic-tac-toe project, the element `id` attribute values were used to select individual elements in the DOM. We're using React instead manipulating the DOM directly, so we won't need those element `id` attributes. We're using [CSS Modules](#) for our game board styles, so we need to avoid hyphens in our CSS class names. We also need to use the `className` attribute instead of `class`. Accounting for all of that, let's update the `TODO` comment for rendering game board squares in the `Game` component to this:

```
<div className='square row_1 col_1'></div>
<div className='square row_1 col_2'></div>
<div className='square row_1 col_3'></div>
<div className='square row_2 col_1'></div>
<div className='square row_2 col_2'></div>
<div className='square row_2 col_3'></div>
<div className='square row_3 col_1'></div>
<div className='square row_3 col_2'></div>
<div className='square row_3 col_3'></div>
```

We need to track when players click on a specific square and determine which square index (`0` through `8`) that they clicked on. To do that, let's define a `Square` function component in your `Game.js` file for rendering squares:


```
const Square = ({ squareIndex, row, col }) => {
  const rowStyleName = `row_${row}`;
  const colStyleName = `col_${col}`;

  const handleClick = () => {
    console.log(`Clicked on square index: ${squareIndex}...`);
  }

  return (
    <div
      onClick={handleClick}
      className={` ${styles.square} ${styles[rowStyleName]} ${styles[colStyleName]}
        { /* TODO Render square "X" or "O" image */ }
    </div>
  );
};
```

Notice how the component accepts `squareIndex`, `row`, and `col` props to determine what index this square represents and to render the correct CSS Module class names.

Now we can use the `Square` component in the `Game` component's `render` method. Replace each `<div>` with the `.square` class to be a `<Square>` component taking in a `squareIndex` prop, a `row` prop, and a `col` prop. For reference, here are the entire contents of the `./src/components/Game.js` file:

```
// ./src/components/Game.js

import React from 'react';
import styles from './Game.module.css';

const Square = ({ squareIndex, row, col }) => {
  const rowStyleName = `row_${row}`;
  const colStyleName = `col_${col}`;

  const handleClick = () => {
    console.log(`Clicked on square index: ${squareIndex}...`);
  }

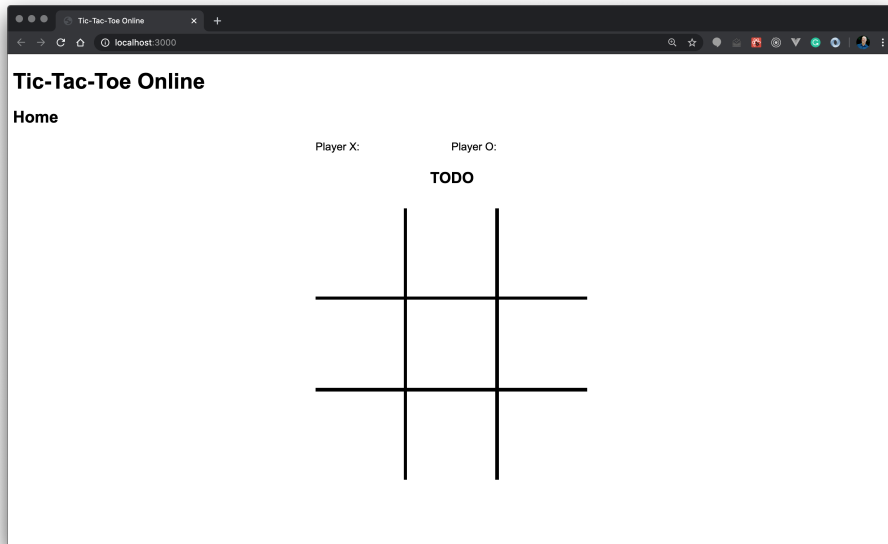
  return (
    <div
      onClick={handleClick}
      className={` ${styles.square} ${styles[rowStyleName]} ${styles[colStyleName]}
        { /* TODO Render square "X" or "O" image */ }
    </div>
  );
};

const Game = () => {
  return (
    <div className={styles.game}>
      <div className={styles.players}>
        <div>Player X: { /* TODO Render player 1 name */}</div>
        <div>Player O: { /* TODO Render player 2 name */}</div>
      </div>
      <h3 className={styles.announcement}>TODO</h3>
      <div className={styles.tic_tac_toe_board}>
        <Square squareIndex={0} row={1} col={1} />
        <Square squareIndex={1} row={1} col={2} />
        <Square squareIndex={2} row={1} col={3} />
        <Square squareIndex={3} row={2} col={1} />
        <Square squareIndex={4} row={2} col={2} />
        <Square squareIndex={5} row={2} col={3} />
        <Square squareIndex={6} row={3} col={1} />
        <Square squareIndex={7} row={3} col={2} />
        <Square squareIndex={8} row={3} col={3} />
      </div>
    </div>
  );
};
```

```
);
}

export default Game;
```

Run your client application again (`npm start`); this time you should see your empty tic-tac-toe game board displayed:



Phase 4: Prompting users for their player names

Before users start a game of tic-tac-toe, let's prompt them for their player name. Requiring each user to have a player name will make it easier for us to prompt a player to take their turn (i.e. "Select a square {player name}!").

Setting up a simple form

In the `Home` component, add a new state variable named `playerName`. We're using Hooks, so we'll call the `useState` Hook to declare the state variable:

```
const [playerName, setPlayerName] = useState('');
```

Now add a simple form, containing a single `<input>` element and a `<button>` element, to prompt the user for their player name. Let's also change the `<h2>` heading element to "Welcome!" and add a brief welcome message that prompts the user for their player name:

```
<div>
  <h2>Welcome!</h2>
  <p>Please provide your player name and
    click the "Play Game" button to start a game.</p>
  <form onSubmit={onSubmit}>
    <input type='text' value={playerName}
      onChange={onChange} />
    <button>Play Game</button>
  </form>
</div>
```

Notice that the `<form>` and `<input>` elements respectively reference `onSubmit` and `onChange` event handler functions. Go ahead and add those event handler functions. The `onSubmit` event handler function should prevent the form's default submit action and the `onChange` event handler function should use the target `<input>` element's `value` to update the `playerName` state variable.

At this point, you should be able to enter and remove characters in the `<input>` element and click the "Play Game" button, though nothing will occur (as expected since your `onSubmit` function simply prevents the form's default submit action).

Validating the form

To ensure that the user enters a player name before starting a game, let's add validation to our form.

Add another state variable named `errors` to your `Home` component with an initial value of an empty array (i.e. `[]`):

```
const [errors, setErrors] = useState([]);
```

In the `onSubmit` event handler function, declare a variable named `errorsToSet` set to an empty array. If the `playerName` state variable is falsy push a message onto the `errorsToSet` array containing the text "Please provide a player name." Then if the `errorsToSet` array contains an element, use it to set the `errors` state variable by invoking `setErrors`.

When updating state variables that reference objects and arrays, always prefer to update the state variables with new objects or arrays instead of modifying or mutating the existing objects or arrays. For example, instead of pushing an element onto the `errors` state variable in the `onSubmit` event handler function, we're creating a new array, pushing an element onto the new array, and then passing the new array into a call to `setErrors`.

To render the validation errors, you can create a `ValidationErrors` subcomponent in your `Home.js` file like this:

```
const ValidationErrors = ({ errors }) => {
  if (errors === null || errors.length === 0) {
    return null;
  }

  return (
    <div>
      <p>Please correct the following errors:</p>
      <ul>
        { errors.map(error => <li key={error}>{error}</li> ) }
      </ul>
    </div>
  );
};
```

And then add it just above the form passing in the `errors` state variable:

```
<ValidationErrors errors={errors} />
<form onSubmit={onSubmit}>
  <input type='text' value={playerName}
    onChange={onChange} />
  <button>Play Game</button>
</form>
```

Passing the player name up to the `App` component

Now that the `Home` component has a form to prompt the user for their player name, we need a way to pass the player name back up to the `App` component so that it can be kept with all of the other global state (that's yet to be defined).

In the `App` component, call the `useState` Hook to declare a `playerName` state variable:

```
const [playerName, setPlayerName] = useState('');
```

Next, declare an `updatePlayerName` function that accepts a `playerName` parameter and calls `setPlayerName` to update the `playerName` state variable:

```
const updatePlayerName = (playerName) => {
  setPlayerName(playerName);
};
```

Then pass the `updatePlayerName` function into the `Home` component as a prop:

```
<Home updatePlayerName={updatePlayerName} />
```

Back in the `App` component, use destructuring to get a reference to the `updatePlayerName` prop and call it within the `onSubmit` event handler function:

```
const Home = ({ updatePlayerName }) => {
  // Code removed for brevity.

  const onSubmit = (e) => {
    e.preventDefault();

    const errorsToSet = [];

    if (!playerName) {
      errorsToSet.push('Please provide a player name.');
```

Lastly, update the `App` component to render the `Game` component if there's a player name, otherwise render the `Home` component:

```
<div>
  <h1>Tic-Tac-Toe Online</h1>
  {playerName ? (
    <Game playerName={playerName} />
  ) : (
    <Home updatePlayerName={updatePlayerName} />
  )}
</div>
```

There are multiple ways to conditionally display elements in JSX. The above example uses an inline expression. Earlier you saw an example of using a subcomponent to conditionally displaying validation messages. Feel free to use the approach that you feel is easiest to write and read.

Testing

Take a moment to test your validation error rendering by submitting an empty player name. You should see your `'Please provide a player name.'` error rendered. Upon a valid form submission to set the `playerName`, your application should be rendering the `<Game>` component instead of `<Home>`. Test that this is working before moving forward.

If you test your client application again, you should see the `Home` component displayed first, prompting you to enter your player name. Click the "Play Game" button without entering a player name to test that you receive a validation error message asking you to enter a player name. Then provide a player name and click the "Play Game" button. You should now see the `Game` component being displayed.

It's worth noting that we haven't done anything to prevent a user from entering a player name that's already been provided by another player. For now, we'll make a point to enter unique player names when testing so that we can turn our attention to setting up the client/server interaction using WebSockets.

Phase 5: Setting up WebSockets

Things are moving nicely along! With the player name available, we're ready to set up the WebSocket server and update the client to connection to the server and send a message.

As you set up the server and client to use WebSockets, you'll notice that the APIs are very similar. Both the server and the client can send and receive messages (that's the "two-way" communication that we're looking for) and both fire events when an errors occur or when connections are closed. An important distinction between the server and the client is that only the server *is listening for new connections* and only the client *can initiate a new connection*.

Update the server

We'll be using the `ws` npm package to set up a WebSocket server, so install it in your server project using npm:

```
npm install ws@^7.0.0
```

In the `app` module (the `app.js` file), import the `ws` module as `WebSocket` :

```
const WebSocket = require('ws');
```

Just after the call to the `createServer` function (i.e.

`const server = createServer(app);`), create a WebSocket server by calling the `WebSocket.Server` method with the `new` keyword:

```
const wss = new WebSocket.Server({ server });
```

Notice that we're passing in the existing HTTP server by setting the `server` variable as a property on an options object. After creating the WebSocket server we can listen for connections by listening for `connection` events:

```
wss.on('connection', (ws) => {  
  });
```

When a WebSocket connection is established, the callback function will be called with the WebSocket passed in via the `ws` parameter. We can then listen for `message` and `close` events on the WebSocket:

```
wss.on('connection', (ws) => {  
  ws.on('message', (jsonData) => {  
  });  
  
  ws.on('close', () => {  
  });  
});
```

`message` events are fired when a message is received from the client while `close` events are fired when the WebSocket connection is closed. For now, just add a `TODO` comment for the `close` event handler callback function:

```
ws.on('close', () => {  
  // TODO Cleanup the player that's associated with this WS.  
});
```

In the `message` event handler callback function, define a parameter named `jsonData` and call a function named `processIncomingMessage` passing in the `jsonData` parameter and the enclosed `ws` parameter (from the `connection` event handler callback function):

```
ws.on('message', (jsonData) => {  
  processIncomingMessage(jsonData, ws);  
});
```

The `jsonData` parameter is set to the data for the incoming message which will be formatted as JSON (we'll see how to do that from the client in just a bit).

Now declare a `processIncomingMessage` function that logs the `jsonData` to the console (to help with testing and debugging) and uses the `JSON.parse` method to parse the `jsonData` to a JavaScript object:

```
const processIncomingMessage = (jsonData, ws) => {  
  console.log(`Processing incoming message ${jsonData}...`);  
  
  const message = JSON.parse(jsonData);  
};
```

The structure of the WebSocket message is completely up to us to decide. The WS specification has no opinion about the structure of the message payload. For this application, on both the client and server, let's use the following message structure:

```
{  
  type: 'the-message-type',  
  data: {  
    // One or more properties for the message data.  
  },  
}
```

Using the above general message structure, when the client sends a message to add a new player, the message will look like this:

```
{  
  type: 'add-new-player',  
  data: {  
    playerName: '[the player name]',  
  },  
}
```

After parsing the JSON formatted data to an object, we can switch on the `message.type` property to process specific message types:

```
const processIncomingMessage = (jsonData, ws) => {
  console.log(`Processing incoming message ${jsonData}...`);

  const message = JSON.parse(jsonData);

  switch (message.type) {
    case 'add-new-player':
      addNewPlayer(message.data.playerName, ws);
      break;
    default:
      throw new Error(`Unknown message type: ${message.type}`);
  }
};
```

Notice how we're throwing an error if the message type is an unexpected or unknown message type. This will help us when testing and debugging if something goes wrong with the client message type.

For now, just stub out the `addNewPlayer` function:

```
const addNewPlayer = (playerName, ws) => {
  // TODO Handle adding the new player.
};
```

Updating the client

Before we can test the WebSocket server, we need to update the client to create a WebSocket connection and send a message to the server.

To start, add an `.env` file to the root of the `client` folder with the following contents:

```
REACT_APP_WS_URL=ws://localhost:8080
```

Notice that we use `ws` instead of `http` to specify the WebSocket URL. The `localhost:8080` hostname and port is the Express server that's hosting the

WebSocket server. Defining an environment variable will make it easier for you later on to set the WebSocket URL to the correct value for each environment.

If you were using HTTPS (SSL/TLS) for your HTTP traffic, you'd need to use `wss` instead of `ws` to indicate that you want to make a secure WebSocket connection. Failing to do that would result in a browser error.

At the top of the `App` module, import two additional Hooks, `useEffect` and `useRef`:

```
import React, { useState, useEffect, useRef } from 'react';
```

The `useEffect` Hook give us a way to add code to function components that will cause side effects. We'll put all of the code that's responsible for creating and configuring the WebSocket in a `useEffect` Hook as the WebSocket will cause side effects as it sends messages to the server.

The `useRef` Hook gives us a convenient way to store a reference to an object that will persist for the full lifetime of the component. We'll use it in just a bit to store away the WebSocket object so that we can interact with it later on.

Inside of the `App` component, just after the call to the `useState` Hook to declare the `playerName` state variable, call the `useRef` Hook to declare a `websocket` variable:

```
const websocket = useRef(null);
```

Then call the `useEffect` Hook and pass in an arrow function:

```
useEffect(() => {

});
```

By default, the function passed into the `useEffect` Hook (referred to as the "effect") will run after every completed render. We can change the default behavior by passing in a second argument that's an array of values that the effect depends on:

```
useEffect(() => {  
  
}, [playerName]);
```

Now our effect will only run when the `playerName` state variable is changed. We can add an additional optimization by immediately returning from the function if the `playerName` variable doesn't have a value (for our particular use case, it doesn't make any sense to create a WebSocket if we don't have a `playerName` value):

```
useEffect(() => {  
  if (!playerName) {  
    return;  
  }  
  
}, [playerName]);
```

Within the effect, create a new client-side WebSocket object by passing in the URL of the WebSocket server, represented by the `REACT_APP_WS_URL` environment variable. Then set the `websocket` ref object's `current` property to an object literal with a `ws` property for the WebSocket:

```
useEffect(() => {  
  if (!playerName) {  
    return;  
  }  
  
  const ws = new WebSocket(process.env.REACT_APP_WS_URL);  
  
  // TODO Define event listeners.  
  
  websocket.current = {  
    ws,  
  };  
}, [playerName]);
```

Setting the `websocket` ref object's `current` property to an object literal (instead of the WebSocket object directly) gives us a safe, convenient way to add references to inline helper functions (we'll write one later in this project).

It's worth noting that updating or changing the `websocket` ref object's `current` property won't cause the component to render. The ref object's `current` property is similar in function to an ES2015 class instance field.

We can also return a cleanup function from our effect. This function will be called before the effect is ran so that the previous execution of the effect can be properly cleaned up. To cleanup our effect, we need to call the `close` method on the WebSocket object (if it's available) to close the connection to the server:


```

useEffect(() => {
  if (!playerName) {
    return;
  }

  const ws = new WebSocket(process.env.REACT_APP_WS_URL);

  // TODO Define event listeners.

  websocket.current = {
    ws,
  };

  return function cleanup() {
    if (websocket.current !== null) {
      websocket.current.ws.close();
    }
  };
}, [playerName]);

```

The WebSocket object provides four events:

- `open` - Fires when the connection is opened;
- `message` - Fires when a message is received;
- `error` - Fires when the connection has been closed because of an error; and
- `close` - Fires when the connection is closed.

We can listen for these events by assigning an event listener to the following properties. Replace the `TODO` comment for defining event listeners to the WebSocket event listeners below:

```

useEffect(() => {
  if (!playerName) {
    return;
  }

  const ws = new WebSocket(process.env.REACT_APP_WS_URL);

  ws.onopen = () => {
  };

  ws.onmessage = (e) => {
    console.log(e);
  };

  ws.onerror = (e) => {
    console.error(e);
  };

  ws.onclose = (e) => {
    console.log(e);
  };

  websocket.current = {
    ws,
  };

  return function cleanup() {
    if (websocket.current !== null) {
      websocket.current.ws.close();
    }
  };
}, [playerName]);

```

There are two ways to assign event listeners: using the above properties (i.e. `onopen` , `onmessage` , `onerror` , or `onclose`) or using the `addEventListener` method and passing in the event name and a callback function (i.e. `ws.addEventListener('message', (e) => console.log(e));`). Either approach is valid; use the one that you or your team prefers.

When the WebSocket connection is opened, send an `add-new-player` message to the server with the `playerName` value as the message data:

```
ws.onopen = () => {
  const message = {
    type: 'add-new-player',
    data: {
      playerName,
    },
  };

  ws.send(JSON.stringify(message));
};
```

The message structure for the outgoing message aligns with what we described earlier when we set up the WebSocket server. Notice how you are manually structuring the `message` object to generate the message structure below. Also notice that the `JSON.stringify` method is used to format the message as JSON (as the server is expecting it to be) before passing it to the WebSocket `send` method.

```
{
  type: 'add-new-player',
  data: {
    playerName: '[the player name]'
  }
}
```

Testing

Start your server by running `npm start` from a terminal within the `server` folder then start the client by running `npm start` within the `client` folder. In the client, enter a player name and click the "Play Game" button. In the server's terminal window, you should see something similar to the following output:

```
Processing incoming message {"type":"add-new-player","data":{"playerName":"[the p
```

Congrats! You just created a WebSocket server, initiated a WebSocket connection from a React application, and received the WebSocket message on the server.

Notice that we don't have set up CORS to give the client application access to the WebSocket server that's running on a different `localhost` port. WebSocket connections aren't restricted to same-origin like HTTP requests are.

Phase 6: Starting a game

Once two players have connected to the server using WebSockets, we're ready to start a game. To do that, we need define a couple of classes to track player and game state data on the server.

Tracking player and game state data on the server

Add a `game-state.js` file to the root of the server project. Then define two classes in the module:

- A `Player` class to track connected players; and
- A `Game` class to encapsulate the logic and state for a game of tic-tac-toe.

```

class Player {
  constructor(playerName, ws) {
    this.playerName = playerName;
    this.ws = ws;
  }

  getData() {
    return {
      playerName: this.playerName,
    };
  }
}

```

```

class Game {
  constructor(player1) {
    this.player1 = player1;
    this.player2 = null;
    this.player1Symbol = 'X';
    this.player2Symbol = 'O';
    this.currentPlayer = player1;
    this.squareValues = ['', '', '', '', '', '', '', '', ''];
    this.gameOver = false;
    this.winner = null;
    this.statusMessage = null;
  }

  getPlayers() {
    return [this.player1, this.player2];
  }

  getData() {
    return {
      player1: this.player1.getData(),
      player2: this.player2.getData(),
      player1Symbol: this.player1Symbol,
      player2Symbol: this.player2Symbol,
      currentPlayer: this.currentPlayer.getData(),
      squareValues: this.squareValues,
      gameOver: this.gameOver,
      winner: this.winner ? this.winner.getData() : null,
      statusMessage: this.statusMessage,
    };
  }
}

```

Be sure to export both classes from the module:

```

module.exports = {
  Game,
  Player,
};

```

The `Player` class is a simple class that's used to associate a player name with a WebSocket connection. The `getData` method is a convenience method that we'll call when creating WebSocket messages to get the data for the player.

The `Game` class encapsulates our game state on the server. When the client app interacts with the server, the game state on the server will be updated. We'll be adding additional methods to this class as we implement functionality in the game. This class also has a `getData` method that'll be used to get the data for the game when creating WebSocket messages.

Ideally, our player and game state data would be persisted to a database, so that if/when the server is restarted, the data would not be lost. To keep things as simple as possible for now, we'll just store the data in memory.

Sending the `start-game` message to the client

Now we can update the `addNewPlayer` function in the `app` module to create a new game when the first player connects and to start the game when the second player connects.

Import the `Game` and `Player` classes into the `app` module (the `app.js` file):

```
const { Game, Player } = require('./game-state');
```

Declare a `game` variable just after the line of code that creates the WebSocket server (i.e. `const wss = new WebSocket.Server({ server });`):

```
let game = null;
```

This module-level global variable is how we'll be persisting the game across WebSocket messages.

Navigate to the `addNewPlayer` function in the `app` module and instantiate an instance of the `Player` class, passing in the `playerName` and `ws` parameters:

```
const addNewPlayer = (playerName, ws) => {  
  const player = new Player(playerName, ws);  
  // TODO  
};
```

Then add an `if / else if / else` statement that does the following:

- if the `game` global variable is `null`, then instantiate an instance of the `Game` class, passing in the new player (who becomes player "1")
- else if the game's `player2` property is `null`, then set the game's `player2` property to the instantiated player and call the `startGame` function (we'll define that function is just a bit)
- else log to the console that we're ignoring a player addition (``Ignoring player ${playerName}...``) and `close()` the player's WebSocket connection (more about this in a bit)

Ignoring player additions once a game has been started isn't ideal, but for now, it's a stop gap so that we can focus on implementing the game. In the bonus phases for this project, you'll get a chance to extend the server to support multiple concurrent games.

```
const addNewPlayer = (playerName, ws) => {
  const player = new Player(playerName, ws);

  if (game === null) {
    game = new Game(player);
  } else if (game.player2 === null) {
    game.player2 = player;
    startGame();
  } else {
    // TODO Ignore any additional player connections.
    console.log(`Ignoring player ${playerName}...`);
    ws.close();
  }
};
```

Once the `game` global variable holds a reference to an instance of the `Game` class and its `player1` and `player2` properties are both set to instances of the `Player` class, we can use the players' WebSocket connections to broadcast a message containing the current game state. To do that, define a function named `startGame` that:

- Calls the `Game` `getData` method to get the data for the current game state;
- Sets the `data.statusMessage` property to a message that prompts the current player to select a square; and
- Calls the `broadcastMessage` function (we'll define this helper function in just a bit) to send a `start-game` message to both players.

```
const startGame = () => {
  const data = game.getData();
  data.statusMessage = `Select a square ${game.currentPlayer.playerName}!`;
  broadcastMessage('start-game', data, game.getPlayers());
};
```

Remember that the structure of the messages that are sent between the client and the server is completely up to us to define. The structure of the `start-game` message type will look like this (before it's formatted as JSON):

```
{
  type: 'start-game',
  data: {
    player1: {
      playerName: 'Bob',
    },
    player2: {
      playerName: 'Sally',
    },
    player1Symbol: 'X',
    player2Symbol: 'O',
    currentPlayer: {
      playerName: 'Bob',
    },
    squareValues: ['', '', '', '', '', '', '', '', '', ''],
    gameOver: false,
    winner: null,
    statusMessage: 'Select a square Bob!',
  }
}
```

To see where this data structure is defined, see the `Game` and `Player` `getData` methods. In a bit, we'll update the React client to use this data to render the UI for the game.

All that's left to do on the server (at least for now), is to define the `broadcastMessage` helper function!

The `broadcastMessage` function accepts a message `type`, the message `data`, and an array of `Player` class instances. The message `type` and `data` are used to create a simple object literal which in turn is formatted as JSON using the `JSON.stringify` method. To assist with testing and debugging, the message JSON is logged to the console. Then the `players` array is enumerated using the `Array` `forEach` method.

Remember that each `Player` class instance holds a reference to the player's WebSocket connection via the `ws` property. The WebSocket connection object provides a `send` method that when called, sends a message to the connected client. The first argument passed into the `send` method is the message to send and the second argument is a callback function that's called if an error occurs when sending the message. For now, just log any errors to the console.

Your `broadcastMessage` helper function should look something like this:

```
const broadcastMessage = (type, data, players) => {
  const message = JSON.stringify({
    type,
    data,
  });

  console.log(`Broadcasting message ${message}...`);

  players.forEach((player) => {
    player.ws.send(message, (err) => {
      if (err) {
        // TODO Handle errors.
        console.error(err);
      }
    });
  });
};
```

Processing the `start-game` message on the client

Before we update the `App` component to handle the processing of `start-game` messages received from the server, let's update the `Game` component to display a "Waiting for game to start..." message when the `Game` component initially loads. The first player to provide their player name and connect to the server will see this message while they wait for the second player to provide their player name and connect to the server.

Add another `useState` Hook to initialize a `game` state variable to the `App` component just below the existing `useState` Hook that initializes the `playerName` state variable:

```
const [game, setGame] = useState(null);
```

Then pass the `game` state variable into the `Game` component as a prop:

```
<Game playerName={playerName} game={game} />
```

Update the `Game` component to destructure the `playerName` and `game` props. Then have your component use a ternary statement to check the truthiness of the `game` prop to conditionally display the following "Waiting for game to start..." message (instead of the game board) if the `game` prop is falsy:

```
<h3 className={styles.announcement}>Waiting for game to start...</h3>
```

Now let's update the `App` component to process the `start-game` message from the server! Update the WebSocket `onmessage` event listener function to:

- Log the `e.data` property to the console;
- Call the `JSON.parse` method to parse the JSON formatted message data; and
- Switch on the `message.type` property to handle the `start-game` message type.

```
ws.onmessage = (e) => {
  console.log(`Processing incoming message ${e.data}...`);

  const message = JSON.parse(e.data);

  switch (message.type) {
    case 'start-game':
      setGame(message.data);
      break;
    default:
      throw new Error(`Unknown message type: ${message.type}`);
  }
};
```

To process the `start-game` message and update the `game` state variable, we just need to call the `setGame` method passing in the `message.data` property. Updating the `game` state variable will cause React to re-render the `Game` component.

Updating the `Game` component to render the game state

To render the game state, update the `Game` component's JSX to use the following game state properties:

- The `game.player1Symbol`, `game.player2Symbol`, `game.player1.playerName`, and `game.player2.playerName` properties can be used to display the player information above the game board (replace the `Player X: ...` and `Player O: ...` TODO notes).
- The `<h3>` element with a `className` referencing `styles.announcement` can be updated to display the `game.statusMessage` property as content. The `game.statusMessage` property will be used going forward to communicate to the players the current status of the game.
- The `game.squareValues` array can be used to set a `value` prop on each `Square` subcomponent to render either an "X" or "O" when appropriate:

```
<Square squareIndex={0} value={game.squareValues[0]} row={1} col={1} />
<Square squareIndex={1} value={game.squareValues[1]} row={1} col={2} />
<Square squareIndex={2} value={game.squareValues[2]} row={1} col={3} />
<Square squareIndex={3} value={game.squareValues[3]} row={2} col={1} />
<Square squareIndex={4} value={game.squareValues[4]} row={2} col={2} />
<Square squareIndex={5} value={game.squareValues[5]} row={2} col={3} />
<Square squareIndex={6} value={game.squareValues[6]} row={3} col={1} />
<Square squareIndex={7} value={game.squareValues[7]} row={3} col={2} />
<Square squareIndex={8} value={game.squareValues[8]} row={3} col={3} />
```

To render the "X" and "O"s in the game board squares, download the following SVG files:

- [playerX.svg](#)
- [playerO.svg](#)

Then add a new folder named `assets` to the `src` folder and copy the SVG files into new folder.

Back in the `Game` module, import the SVG files at the top of the module:

```
import playerX from '../assets/playerX.svg';
import playerO from '../assets/playerO.svg';
```

Just above the definition for the `Square` subcomponent, define a `SquareImage` subcomponent that'll render a square's image using an `` element:

```
const SquareImage = ({ value }) => {
  if (value === '') {
    return null;
  } else if (value === 'X') {
    return <img src={playerX} alt='X' />
  } else {
    return <img src={playerO} alt='O' />
  }
};
```

Notice that we can simply set the `` element's `src` attribute to the imported `playerX` or `playerO` SVG files. This is possible because of the front-end build process provided by Create React App. Create React App configures webpack with support for loading images (as well as CSS, fonts, and other file types) which allows you to add an image file to your project, import it directly into a module, and render it in a React component.

Now we can update the `Square` subcomponent. Add a `value` prop and render the `SquareImage` component inside of the `<div>` element:

```
const Square = ({ squareIndex, value, row, col }) => {
  const rowStyleName = `row_${row}`;
  const colStyleName = `col_${col}`;

  const handleClick = () => {
    console.log(`Clicked on square index: ${squareIndex}...`);
  }

  return (
    <div
      onClick={handleClick}
      className={` ${styles.square} ${styles[rowStyleName]} ${styles[colStyleName]} `
    >
      <SquareImage value={value} />
    </div>
  );
};
```

Handling closed connections

Before we test our latest changes, let's handle closed connections on both the server and the client. If we don't handle closed connections, it'd be very easy for the state to get out of sync between the server and the client.

On the client, handling closed connections is relatively straightforward. Update the `onclose` event listener function to log a message that the connection closed, reset the `websocket` ref object, and reset the `playerName` and `game` state variables:

```
ws.onclose = (e) => {
  console.log(`Connection closed: ${e}`);
  websocket.current = null;
  setPlayerName('');
  setGame(null);
};
```

Resetting the ref object and state variables will return the user back to the initial application state which results in the player being prompted again for their player name. This is essentially throwing up our hands and declaring "images/oh no... something went wrong... let's try that again". Not ideal, but for now, it'll allow us to keep our focus on finishing the initial implementation of the game.

On the server, update the WebSocket `close` event handler callback function to this:


```
ws.on('close', () => {
  // If there's a game available...
  if (game !== null) {
    const { player1, player2 } = game;

    // If the closed WS belonged to either player 1 or player 2
    // then we need to abort the game.
    if (player1.ws === ws || (player2 !== null && player2.ws === ws)) {
      // If the closed WS doesn't belong to player 1
      // then close their WS, otherwise if there's a
      // player 2 then close their WS.
      if (player1.ws !== ws) {
        player1.ws.close();
      } else if (player2 !== null) {
        player2.ws.close();
      }
      game = null;
    }
  }
});
```

Handling closed connections on the server is a little trickier than it is on the client. On the server we need to determine if there's an active game, and if there is we need to check if the closed connection belonged to either player 1 or player 2 (if player 2 is available).

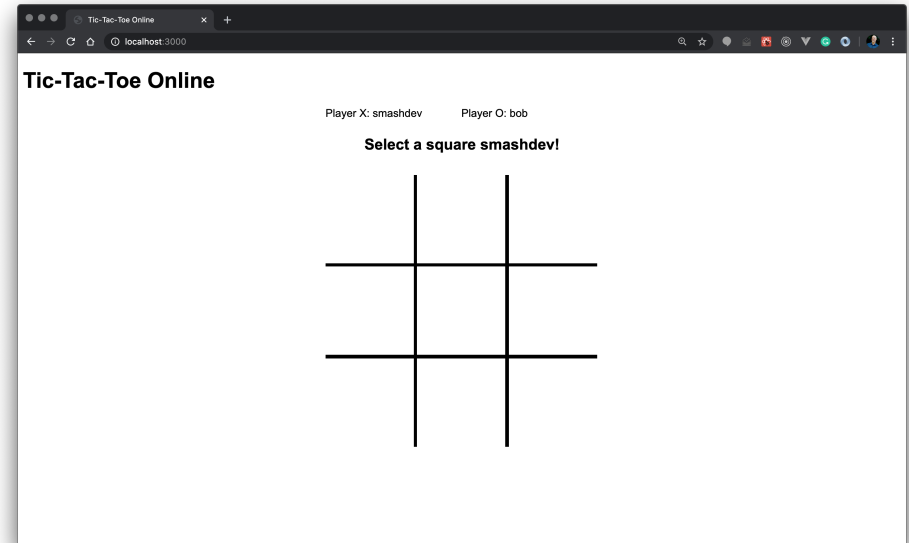
Testing

To test the changes to the server and the client, start both the server and the client. Then, in the browser, provide a player name for the first player. You should see the message "Waiting for game to start..." displayed in the browser.

In the terminal window where you started the server, you should see the following message:

```
Processing incoming message {"type":"add-new-player","data":{"playerName":"[the p
```

Now open a second browser tab and browse to `http://localhost:3000` and enter a player name for the second player. This time, you should see the game board displayed in the browser.



In the server terminal window, you should see the following messages:

```
Processing incoming message {"type":"add-new-player","data":{"playerName":"[the p
Broadcasting message {"type":"start-game","data":{"player1":{"playerName":"[playe
```

Phase 7: Supporting player turns

With the game started, we can turn our attention towards supporting player turns, so that they can select squares on the game board! Clicking a game board square will result in three things happening:

- The client will handle the game board square click in the `Game` component and send a `select-game-square` message to the server;
- The server will receive and process the `select-game-square` message, check the status of the game to see if the game has ended, and send an `update-game` or `end-game` message back to the client; and
- The client will receive the `update-game` or `end-game` message and re-render the `Game` component with the new game state.

Now that the general WebSocket plumbing is in place to support the two-way communication between the server and client, the remaining phases in this project will let you apply what you've learned with less guidance.

Updating the client to handle game board square clicks

Add an additional prop named `selectGameSquare` to the `Game` component (we'll set this prop from within the `App` component in just a bit):

```
const Game = ({ playerName, game, selectGameSquare }) => {
  // Code removed for brevity.
}
```

To handle game board square clicks, define a function within the `Game` component named `selectSquare` that:

- Accepts a `squareIndex` parameter;
- Immediately returns if the game is over or if the player is not the current player (see the `game.gameOver` and `game.currentPlayer.playerName` properties); and
- Calls the `selectGameSquare` prop and passing in the `squareIndex` parameter.

In the `Game` components JSX, pass the `selectSquare` function into each of the `<Square>` components using a prop of the same name:

```
<Square squareIndex={0} value={game.squareValues[0]} row={1} col={1} selectSquare={/* Other components removed for brevity. */}
```

In the `Square` subcomponent, update the prop destructuring to get a reference to the `selectSquare` prop. Then call the `selectSquare` prop from within the `handleClick` event handler function passing in the `squareIndex` prop value.

In the `App` component, we need to handle the square selection and send a `select-game-square` message to the server. Start by defining a function named `selectGameSquare` that accepts a `squareIndex` parameter and sends a `select-game-square` message to the server that looks like this (before its converted to JSON):

```
{
  type: 'select-game-square',
  data: {
    squareIndex: 0,
  },
}
```

To see an example of how to send a message to the server, look for where the `ws.send` method is being called. To help make it a bit easier to send messages to the server (and to keep our code DRY), define a helper function named `sendMessage` within the effect that creates the WebSocket object. Add a property to the object literal that's assigned to the `websocket` ref object's `current` property so that you can call it from elsewhere in the component (i.e. `websocket.current.sendMessage('select-game-square', { squareIndex })`):

```
const sendMessage = (type, data) => {
  // TODO Create and send the message to the server.
};

websocket.current = {
  ws,
  sendMessage,
};
```

The `sendMessage` function should accept `type` and `data` parameters and combine them into an object literal to create our self-imposed WebSocket message structure:

```
{
  type: 'the-message-type',
  data: {
    // One or more properties for the message data.
  },
}
```

Use the `JSON.stringify` method to convert the message to JSON, then log the message to the console (to assist with testing and debugging), and passing the JSON message into a call to the `ws.send` method. Now update the `onopen` event listener function to make use of the `sendMessage` function:

```
ws.onopen = () => {
  sendMessage('add-new-player', { playerName });
};
```

Be careful passing the argument for the `sendMessage` `data` parameter. Think about the shape of the `data` property for each message type. If you get the message structure wrong, the server will start to throw errors when receiving messages from the client, so keep an eye out for that.

To wrap up this part of the updates, add a `selectGameSquare` prop on the `Game` component and set it to a reference to the `selectGameSquare` function:

```
<Game playerName={playerName} game={game} selectGameSquare={selectGameSquare} />
```

Testing the changes so far

Start the client and the server and open an additional browser tab so that you can enter two player names. In the player 1 browser tab, click a game board square. In the browser developer tools console you should see output confirmation that a message was sent to the server:

```
Sending message {"type":"select-game-square","data":{"squareIndex":0}}...
```

In the server terminal window you should see that an "unknown message type" error has occurred:

```
Processing incoming message {"type":"select-game-square","data":{"squareIndex":0}}
[path to top-level project folder]/solution/server/app.js:75
  throw new Error(`Unknown message type: ${message.type}`);
    ^

Error: Unknown message type: select-game-square
    at processIncomingMessage ([path to top-level project folder]/solution/server
    at WebSocket.<anonymous> ([path to top-level project folder]/solution/server/
    at WebSocket.emit (events.js:305:20)
    at Receiver.receiverOnMessage ([path to top-level project folder]/solution/se
    at Receiver.emit (events.js:305:20)
    at Receiver.dataMessage ([path to top-level project folder]/solution/server/n
    at Receiver.getData ([path to top-level project folder]/solution/server/node_
    at Receiver.startLoop ([path to top-level project folder]/solution/server/nod
    at Receiver._write ([path to top-level project folder]/solution/server/node_m
    at doWrite (_stream_writable.js:464:12)
```

Stop and restart the server and enter your player names again. In the player 2 browser tab, make sure that when you click a game board square a message isn't sent to the server. To confirm that a message wasn't sent, you shouldn't see any

output in the browser developer tools console and you shouldn't see any errors in the server terminal window.

Updating the server to process `select-game-square` messages

As the server error indicated while testing, we need to update the server so that it can process `select-game-square` messages.

To start, update the `processIncomingMessage` function by adding a `case` statement to the `message.type` switch statement so that it can handle `select-game-square` values. Within that `case` statement, call the `selectGameSquare` function passing in the `message.data.squareIndex` property and the enclosing function's `ws` parameter.

Next, define three new functions: `endGame` (not a reference to the Avengers movie), `updateGame`, and `selectGameSquare`:

```
const endGame = () => {
  // TODO
};

const updateGame = () => {
  // TODO
};

const selectGameSquare = () => {
};
```

We'll implement the `endGame` and `updateGame` functions in a bit, so for now, just add `TODO` comments in their function bodies as a reminder.

For the `selectGameSquare` function implementation:

- Define two parameters, `squareIndex` and `ws`;

- Call the `game.getPlayers` method to get the players for the game and use the `ws` parameter to determine which player is selecting the square (*hint*: remember that the `Player` class contains a property that holds a reference to the player's WebSocket connection);
- After determining the player that's taking a turn (i.e. selecting a square), call a new method on the `game` object, `selectSquare`, passing in the player and the `squareIndex` parameter;
- After selecting the square, call another new method on the `game` object, `checkGameStatus`, to determine if the game has ended (either as a win for a player or in a draw);
- If the game has ended, call the `endGame` method and set the `game` variable to `null` (setting the `game` variable to `null` allows a new game to be started); and
- If the game hasn't ended, call the `updateGame` method.

In the `Game` class (located in the `game-state` module), we have two new methods to implement: `selectSquare` and `checkGameStatus`.

For the `selectSquare` method implementation:

- Define two parameters, `player` and `squareIndex`;
- As a bit of defensive coding, immediately return from the function if the selected square index value (i.e. `this.squareValues[squareIndex]`) is not an empty string (i.e. `''` or `''`);
- Using the `this.player1` and `this.player2` properties, determine if the `player` parameter represents player "1" or player "2", then set the selected square index to that player's symbol using either the `this.player1Symbol` or `this.player2Symbol` property; and
- Now that the selected square index has been set the current player's symbol, it's time to end the current player's turn by updating the `this.currentPlayer` property to reference the other player.

For the `checkGameStatus` method implementation:

- Use the `this.squareValues` property, which is an array of the game board's square values, to determine if either player has won the game (be sure to check each row, column, diagonal) or if the game has ended in a draw;
 - There are *many* ways to iterate over the array of square values to determine if a player has won the game;
 - Remember that you built a tic-tac-toe game in an earlier project, so feel free to reference that project for a solution to this coding problem;
- If the game is over (either in a win or draw), set the `this.gameOver` property to `true`;
- Set the `this.winner` property to the player that won the game (if the game ended in a draw, leave the property set to `null`); and
- Optionally return the `this.gameOver` property value from the function so that it can be called from within a conditional statement expression.

Back in the `app` module, we can turn our attention back to implementing the `endGame` and `updateGame` functions.

For the `endGame` function implementation:

- Call the `game.getPlayers` method to get an array of the players;
- Call the `game.getData` method to get the game state data;
- Set the `data.statusMessage` property to an appropriate message to indicate the end of the game (e.g. "Winner: [player name]" or "Winner: Draw!");
 - Consider adding a [class getter function](#) named `gameOverMessage` to the `Game` class that returns a string for the `data.statusMessage`; and
- Send an `end-game` message to each of the players by calling the `broadcastMessage` function passing in the string literal `'end-game'`, the data returned by the `game.getData` method, and the array of players returned by the `game.getPlayers` method.

For the `updateGame` function implementation:

- Call the `game.getPlayers` method to get an array of the players;

- Call the `game.getData` method to get the game state data;
- Set the `data.statusMessage` property to an appropriate message to prompt the next player to take their turn (i.e. "Select a square [current player name]!"); and
- Send an `update-game` message to each of the players by calling the `broadcastMessage` function passing in the string literal `'update-game'`, the data returned by the `game.getData` method, and the array of players returned by the `game.getPlayers` method.

That's a lot of coding! Pat yourself on the back; you just completed the server part of the project!

Updating the client to process `update-game` and `end-game` messages

Now it's time to complete the client part of the project by updating it to process `update-game` and `end-game` messages.

In the `App` component's effect function that creates the `WebSocket` object, update the `onmessage` event listener function to process `update-game` and `end-game` message types by updating the `game` state variable. To do this, call the `setGame` function and pass in the `message.data` property.

Updating the `game` state variable will cause the `Game` component to re-render. When it does, we need to display two buttons within the `Game` if the game has ended (i.e. the `game.gameOver` property is set to `true`):

- A "Play Again" button that when clicked allows the user to play another game;
 - To play another game, set the `game` state variable in the `App` component to `null` and call the `sendMessage` function to send an `add-new-player` message (be sure to pass the player's name for the message's data); and
- A "Quit" button that when clicked resets all of the local state in the client;
 - A convenient way to reset all of the local state is to set the `playerName` state variable to an empty string (i.e. `''` or `""`). Doing this results in

the effect's `cleanup` function being called which will close the WebSocket connection.

For the layout of the buttons, render the following JSX below the game board:

```
{ game.gameOver && (  
  <div className={styles.actions}>  
    <button onClick={playAgainClick}>Play Again</button>  
    <div className={styles.spacer}></div>  
    <button onClick={quit}>Quit</button>  
  </div>  
)}
```

Testing

Everything is in place now to play a complete game of tic-tac-toe! Start the client and the server and run through (at a minimum) the following testing scenarios:

- Test that player 1 can win a game;
- Test that player 2 can win a game;
- Test that completing any row, column, or diagonal wins the game;
- Test that a game can be played to a draw;
- Test that you can choose to play again at the conclusion of a game; and
- Test that you can quit at the conclusion of a game.

Excellent job using WebSockets to create an online version of tic-tac-toe!

Bonus Phases

Now that you've built a *basic* version of the tic-tac-toe game, there are a lot of ways that you could extend this application.

Adding Redux or Context

The state needs for the React application were relatively simple, so we didn't use Redux or Context. But that doesn't mean that you can't still add either one for additional practice.

Support multiple concurrent games

After two players have connected to the server and a game is started, all subsequent WebSocket connections are ignored (i.e. closed immediately after they're opened). Ideally, you'd continue to add new games as players are connected to the server. To do this, you'll need a way to track multiple game instances on the server and to cleanup those instances when a game has been completed.

Player name validation

Currently, when adding a new player, the player name isn't validated to ensure that it's unique. Ideally, when a user submits a player name that's already being used in an active game, the server would return a message to indicate to the client that the supplied player name is already in use so the user could provide a different player name.

Connection health checks

In production, servers can be aggressive about closing inactive WebSocket connections. In some environments, WebSocket connections might be closed after only 60 seconds of inactivity. To keep connections alive, you can add ping/pong health checks to the WebSocket server.

The WebSockets protocol supports the idea of a server sending a "ping" message to a client, and if the client is still connected, it'll send a "pong" message back to the server. For details on how to do this with the `ws` npm package, see [this example in the official documentation](#).

Technically speaking, there's no reason why a client couldn't also send "ping" messages to the server (instead of waiting for the server to send a "ping") but the browser's WebSocket API doesn't currently support sending "ping" messages.

Adding a player lobby

Instead of immediately associating new players with games, you could place them into a player "lobby". If the server arbitrarily limited the number of active games to a small number of games (1-3), the server could add the first two players in line to a game upon the conclusion of one of the active games. Alternatively, you could allow players in the lobby to challenge another player in the lobby. There are lots of ways to implement a lobby... have fun with it!

Adding database persistence

Instead of keeping player and game state in memory on the server, you could persist both to a PostgreSQL database. This would allow the server to be restarted without losing all of the data. You could also implement win/loss/draw history for each player.

Deploying to production

Deploying applications into a new environment can often be challenging. As with most things in life, practice helps, so deploy your Tic-Tac-Toe Online game into a cloud platform like [Heroku](#).

React Solo Project Expectations

Next week, you get a full week to write a compelling front-end application to go into your portfolio.

Expectations

The following is the list of expectations that must be met for a project to be considered complete.

1. Project will be picked from the list of projects from the previous reading or come up with your own idea.
2. You should have the following documentation either in the Wiki of your GitHub repo *or* in a documentation folder in your root directory:
 - Schema
 - Feature Document outlining MVP features as well as additional features (user stories and acceptance criteria *strongly* encouraged)
3. A README with instructions on how to launch the application
4. Your app must have user authentication
 - Sign Up
 - Login / Logout
 - Demo User Login
5. Must have at least one major CRUD feature in addition to user authentication
6. Front-end:
 - React
 - Redux
 - Hooks are optional (but encouraged)
7. Backend:
 - Express
 - Sequelize

Student Responsibilities

1. By EOD today, create your project repo, upload your feature list to your repo and send your project advisor your list of features for sign-off
2. Join your new "circle" slack which you will be invited to by the end of day.
Your circle consists of all the students that have the same project advisor as you. You will be having stand up and end of day chats with this circle.
3. Every day you will attend stand up with your advisor and the other students in your circle -- be prepared to discuss what you worked on the day before and have a plan for what you want to accomplish that day.
4. Complete all nightly reports as per usual and use these reports to reflect on your progress. We rely on these reports to check in on your progress so please, as always, take them seriously. Some examples:
 - "I expected to finish the back- and frontend for logging in, but still need to debug my onSubmit."
 - "At the beginning of the day, I was very confused about the Redux cycle, but going through debugger waterfalls while testing my action creators on the window really helped."
 - "Today I really struggled with destructuring my state. I think the next time I do a project like this, I will structure my frontend state to use more slices and less nesting."

Question Asking

During this project, you will get practice expressing yourself when running into trouble without just saying "it doesn't work". In a professional environment, knowing when to ask for help and not spending too much time on any one problem is critical, but making the most out of the time with your manager or Senior Engineers is also vital.

Questions should be asked in Slack to your entire circle. Before asking a question, please have the following available:

- What you are working on

- A description of the problem
- What the error message says on the server/frontend console (if there is one)
- A relevant code snippet
- The debugging process you've done so far (MANDATORY)

This process allows your fellow students to also help you out -- further, it will allow us to know if multiple students are having similar issues and if so we can have a group session to clarify if necessary.

React Project

You've spent the last two weeks on a whirlwind tour of React and some of its most popular features and libraries in its ecosystem. Now, you get to do something substantial with it while its fresh in your minds.

Next week, you get a full week to write a compelling front-end application to go into your portfolio. It should be something that you can display to demonstrate the skills that you've picked up over the past two weeks. While you do have to use React, the rest is up to you, but we recommend that you use Hooks in your project because those are the latest way that people build things in React and will be most appealing to people that are looking at your project.

Because this is an individual project, you have full freedom (and ultimately responsibility) of being able to get something done. If you want to build the entire fullstack application yourself, that's great! Just make sure you scope the amount of work into something that you feel is best for your week's worth of time. Use your newly-found knowledge of how long it took you to build your other project to determine how much you could get done on your own.

If you choose not to build a full-stack project, but just want to spend all of your time on the front-end, consider building a front-end for *any* of the

group projects, it doesn't have to be your own. Or, you can check out [Programmable Web](#) which lists thousands of APIs that you can use in your project.

Now that you have an understanding of how Websockets work, use the [socket.io](#) library on your backend to make Websocket development easy.

Moreover, search for *third-party React libraries* that you can use to help speed your development or give your application a polished look. Here are some suggestions for you to look at as a start:

- [Grommet](#): A react-based framework that provides accessibility, modularity, responsiveness, and theming in a tidy package
- [Material UI](#): A library of components that adhere to the Material design specifications
- [React Bootstrap](#): A library of React components that wrap Bootstrap
- [Semantic UI React](#): A development framework that helps create beautiful, responsive layouts using human-friendly HTML.

Those are just starting points. You can look for more by using your favorite search engine and terms like "react libraries".

Homework

Think about what you'd like to do, poke around [Programmable Web](#), go back and look at all of the group projects, look at the list of projects at the end of this article. Find something that you would like to do. You already have a strong knowledge of how to build a data API. You have the code to hook in authentication using JWTs. You can reuse your models and migrations from earlier projects.

Planning Day

Put together your proposal. In software development, normally, a week of planning can give you about three months of work. So, your day of figuring out what you'd like your app to do should give you a week

Project ideas

Here is that list of project suggestions, again, with their different requirements. Feel free to do one of them, if you think you have time.

- [AirBnB](#)
 - Spots
 - Bookings
 - Spots search (by location & availability) & Google Maps on search
 - Reviews
 - **Bonus**: Messaging
 - **Bonus**: User/host profiles
- [Asana](#)
 - Projects
 - Tasks
 - Teams
 - Profile for each user
 - **Bonus**: Calendar
 - **Bonus**: Comments on tasks
- [Bandcamp](#)
 - Artist page
 - Song player
 - Search
 - Upload/download songs
 - **Bonus**: Purchase songs
 - **Bonus**: Follows
- [Basecamp](#)
 - To-do Lists

- Basecamp Home View
- Message Board (post questions and allow comments/answers)
- Schedule
- **Bonus:** Upload Documents & Files
- **Bonus:** Messaging within a basecamp
- [BillPin](#)
 - Friends
 - Feed with transactions
 - Owes me/ I owe
 - Split bill
 - **Bonus:** profile
- [BrainScape](#)
 - Create/Delete Decks
 - Study Decks
 - Tags/Categories
 - Search
 - **Bonus:** Badges (associated with progress)
 - **Bonus:** Animation
- [Couchsurfing](#)
 - Spots
 - Bookings
 - Spots search (by location & availability) & Google Maps on search
 - Reviews
 - **Bonus:** Messaging
 - **Bonus:** User/host profiles
- [Discord](#)
 - Servers
 - Channels within servers
 - Live Chat
 - Direct Messaging via private servers
- [Etsy](#)
 - Product Listings

- Shopping Cart
- Comments / Reviews
- Search
- **Bonus:** Categories
- **Bonus:** Favorites
- [Eventbrite](#)
 - Events
 - Registration / Tickets
 - Categories
 - Bookmark events
 - **Bonus:** Google Maps integration
 - **Bonus:** Search
- [Evernote](#)
 - Notes
 - Notebooks
 - Tags
 - Rich-text editing
 - **Bonus:** Reminders
 - **Bonus:** Search
 - **Bonus:** Auto save
- [Feedly](#)
 - Feeds
 - Sources
 - Articles
 - Reads
 - **Bonus:** Favorites
 - **Bonus:** Search
- [Flickr](#)
 - Photos
 - Albums
 - Comments
 - Tags

- **Bonus:** Favorites
- **Bonus:** Follows
- **Genius**
 - Tracks
 - Annotations
 - Comments
 - Upvotes
 - **Bonus:** Tags
 - **Bonus:** Search
- **Goodreads**
 - Books
 - Bookshelves
 - Reviews
 - Read Status (will read, have read, etc.)
 - **Bonus:** Search across multiple models
 - **Bonus:** Tags
- **HipCamp**
 - Spots
 - Bookings
 - Spots search (by location & availability) & Google Maps on search
 - Reviews
 - **Bonus:** Messaging
 - **Bonus:** User/host profiles
- **Indiegogo**
 - Profiles
 - Campaigns
 - Contributions
 - Rewards
 - **Bonus:** Categories
 - **Bonus:** Follows
- **Instructables**
 - Projects

- Commenting on projects
- Adding photos and videos to projects
- Searching projects by keyword
- **Bonus:** Featured project channels
- **Bonus:** Categories
- **Kickstarter**
 - Projects
 - Backing projects & rewards
 - Search
 - Categories / Discover feature
 - **Bonus:** Likes
 - **Bonus:** Credit card payments
- **Medium**
 - Stories
 - Commenting on stories
 - Follows and feed
 - Likes
 - **Bonus:** Topics/categories
 - **Bonus:** Bookmarks
- **Meetup**
 - Groups and joining groups
 - Events and RSVPs
 - Calendar (on group page)
 - Search by location and group info (name, description)
 - **Bonus:** Categories
 - **Bonus:** Calendar (for all groups in search results)
- **Newsblur**
 - RSS feeds
 - Story tagging
 - Sharing on Blurblog/profile
 - Full text search of stories
 - **Bonus:** hide or highlight stories

- **Bonus:** Real-time updating of RSS feeds
- [OpenTable](#)
 - Create and search restaurants
 - Reservations
 - Ratings/reviews
 - Favorites
 - **Bonus:** Discover/explore
 - **Bonus:** points for bookings and usage of site
- [Pinterest](#)
 - Profile
 - Boards and Pins
 - Follows
 - Discover feed on home page
 - **Bonus:** notifications
 - **Bonus:** private boards
 - **Bonus:** likes
- [Pivotal Tracker](#)
 - Projects/Project Page
 - Stories
 - Story Workflow
 - Drag and Drop Prioritization
 - **Bonus:** Iterations
 - **Bonus:** Velocity
- [Poll Everywhere](#)
 - Questions
 - Web URL to take poll
 - Live-update Poll view
 - Group/Ungroup Questions
 - **Bonus:** Text to polls
 - **Bonus:** Reports
- [Product Hunt](#)
 - Products

- Profile Page
- Product Discussion
- Search (Users or Products)
- **Bonus:** Collections
- **Bonus:** Upvotes and Tags
- [Quora](#)
 - Questions
 - Answers/comments on answers
 - Search Questions
 - Topics/Tags
 - **Bonus:** Upvotes, order questions by popularity
 - **Bonus:** Replies to comments
- [Remember the Milk](#)
 - Tasks
 - Lists
 - List summary (time, num tasks, num completed)
 - Search
 - **Bonus:** Autocomplete SmartAdd of task properties
 - **Bonus:** Subtasks
- [Robinhood](#)
 - Dashboard + Portfolio
 - Asset/Stock Detail
 - Watchlist
 - Asset/Stock Search
- [Slack](#)
 - Live chat
 - Channels
 - Direct Message
 - Teams or multi-person DM
 - **Bonus:** Search Messages
 - **Bonus:** Notifications
- [Soundcloud](#)

- Song CRUD
- Playing songs with progress bar with continuous play
- Comments
- User pages
- **Bonus:** Wave Forms
- **Bonus:** Playlists
- **Bonus:** Likes
- [SplitWise](#)
 - Friending
 - Bills
 - Transaction History
 - Comments
 - **Bonus:** Groups
 - **Bonus:** Fake “checkout”
- [Stack Overflow](#)
 - Ask Questions
 - Answer Questions
 - Search for Questions
 - Upvote / Downvote Answer
 - **Bonus:** Question Categories
 - **Bonus:** Comment on Questions / Answers
 - **Bonus:** Polymorphic Up/Down Votes: Questions, Answers, Comments
 - **Bonus:** Code Snippets in Answers
- [Strava](#)
 - Creating Routes
 - Saving Workouts
 - Workout Feed
 - Workout Stats/Totals
 - **Bonus:** Friends
 - **Bonus:** Workout Comments
 - **Bonus:** Social Feed
- [TaskRabbit](#)

- Choose a task
- Provide task details
- Select price & confirm
- Get assigned a tasker
- **Bonus:** Become a Tasker
- **Bonus:** Fake Payment & Billing
- [TeaWithStrangers](#)
 - Choose City
 - Host Event
 - Join Event in your city
 - Dashboard of joined events/hosted events
 - **Bonus:** Google Map API showing events based on location
 - **Bonus:** Suggestions based on event details and user profiles
- [Tumblr](#)
 - Posts form for various post types
 - Feed
 - Follows
 - Likes
 - **Bonus:** Reblog
 - **Bonus:** User show page (blog)
- [Untappd](#)
 - Drinks CRUD
 - Checkins / reviews
 - Review feed
 - Profile
 - **Bonus:** Friendships
 - **Bonus:** Search
 - **Bonus:** Venues
 - **Bonus:** Badges
- [Yammer](#)
 - News feed
 - Groups

- Profile
- Likes
- **Bonus:** Reply
- **Bonus:** Inbox (personal messages)
- **Bonus:** Notifications
- [Yelp](#)
 - Business Page
 - Search / filters
 - Reviews / ratings
 - Map
 - **Bonus:** Mark reviews funny, cool, useful etc.
 - **Bonus:** Profile
 - **Bonus:** Friends
- [Wufoo](#)
 - Build Forms
 - Build various question types
 - Share Forms
 - Display Results
 - **Bonus:** Graph results