INTRO TO THE CORE REDUX API

What is the Redux API?

In this lesson, you will learn how to apply the core concepts of Redux to a real Redux application.

Remember, Redux applications are built upon a one-way flow of data model and are managed by the *store*:

- The *state* is the set of data values that describes the application. It is used to render the user interface (UI).
- Users interact with the UI which dispatch *actions* to the store. An action is an object that expresses a desired change to the state.
- The store generates its next state using a *reducer* function which receives the most recent action and the current state as inputs.
- Finally, the UI is re-rendered based on the new state of the store and the entire process can begin again.

Building an application that follows the core principles of Redux can be done without external libraries. However, the dedicated Redux library provides some very useful tools for handling the most common aspects of building a Redux application and helps ensure that the core Redux principles are enforced.

This lesson will focus on creating a basic Redux application with the `createStore()` method from the Redux API and the following related `store` methods:

- `store.getState()`
- `store.dispatch(action)`
- `store.subscribe(listener)`

Note: The store method `store.replaceReducer(nextReducer)` is an advanced method and will not be covered in this course.

Instructions

Before continuing on to the rest of the lesson, review the contents of the Core Concepts page of the Redux documentation in the connected browser.

To get the most out of the docs, expand the browser window to full screen.

Core Concepts

Imagine your app's state is described as a plain object. For example, the state of a todo app might look like this:

```
{  todos: [{    text: 'Eat food',    completed: true  }, {
text: 'Exercise',    completed: false  }],
visibilityFilter: 'SHOW_COMPLETED'}
```

Copy

This object is like a "model" except that there are no setters. This is so that different parts of the code can't change the state arbitrarily, causing hard-to-reproduce bugs.

To change something in the state, you need to dispatch an action. An action is a plain JavaScript object (notice how we don't introduce any magic?) that describes what happened. Here are a few example actions:

```
{ type: 'ADD_TODO', text: 'Go to swimming pool' }{ type:
'TOGGLE_TODO', index: 1 }{ type: 'SET_VISIBILITY_FILTER',
filter: 'SHOW_ALL' }
```

Copy

Enforcing that every change is described as an action lets us have a clear understanding of what's going on in the app. If something changed, we know why it changed. Actions are like breadcrumbs of what has happened. Finally, to tie state and actions together, we write a function called a reducer. Again, nothing magical about it—it's just a function that takes state and action as arguments, and returns the next state of the app. It would be hard to write such a function for a big app, so we write smaller functions managing parts of the state:

```
function visibilityFilter(state = 'SHOW_ALL', action) {  if
(action.type === 'SET_VISIBILITY_FILTER') {    return
action.filter  } else {    return state  }}
```

```
function todos(state = [], action) {  switch (action.type) {
case 'ADD_TODO':      return state.concat([{ text:
action.text, completed: false }])    case 'TOGGLE_TODO':
return state.map((todo, index) =>        action.index ===
index          ? { text: todo.text, completed:
!todo.completed }          : todo      )    default:
return state  }}
```
Copy

And we write another reducer that manages the complete state of our app by calling those two reducers for the corresponding state keys:

```
function todoApp(state = {}, action) {  return {    todos:
todos(state.todos, action),    visibilityFilter:
visibilityFilter(state.visibilityFilter, action)  }}
```
Copy

This is basically the whole idea of Redux. Note that we haven't used any Redux APIs. It comes with a few utilities to facilitate this pattern, but the main idea is that you describe how your state is updated over time in response to action objects, and 90% of the code you write is just plain JavaScript, with no use of Redux itself, its APIs, or any magic.

Install the Redux Library

The core concepts of Redux are closely tied to a framework known as Flux. Both share the same concept of a one-way flow of data and a centralized store to reduce actions into the application's next state. While Flux was designed as a general pattern which one could follow to build applications, Redux is a library that provides concrete methods to help implement the framework.

To make use of the Redux package, it can be installed using the **N**ode **P**ackage **M**anager (`npm`). Then, its methods can be imported.

Let's start by installing the `redux` package and importing its `createStore()` method. In the next exercise you'll learn how to use this method.

Instructions

1.

In the bash terminal enter the following command for installing the `redux` package using NPM.

```
npm install redux
Checkpoint 2 Passed
```

2.

In the **store.js** file use ES6 import syntax and import the `createStore` method from the `redux` package:

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

store.js

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

bash

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

Create a Redux Store

As you know, every Redux application uses a reducer function that describes which actions can update the state and how those actions lead to the next state.

For example, suppose you wanted to build an application for a light switch. Its reducer might look like this:

```
const initialState = 'on';
const lightSwitchReducer = (state = initialState, action) =>
{
  switch (action.type) {
    case 'toggle':
      return state === 'on' ? 'off' : 'on';
    default:
      return state;
  }
}
```

This reducer handles a single action type `'toggle'` and returns the next state of the store: `'on'` if it had been `'off'` and vice-versa. If an unrecognized action is received, the current state of the store is returned.

The programmer could manually execute the reducer with the current state of the store and the desired action to perform like so:

```
let state = 'on';
state = lightSwitchReducer(state, { type: 'toggle' });
console.log(state); // Prints 'off'
```

However, this is the main responsibility of the `store`. The `store` is an object that enforces the one-way data flow model that Redux is built upon. It holds the current state inside, receives action dispatches, executes the reducer to get the next state, and provides access to the current state for the UI to re-render.

Redux exports a valuable helper function for creating this `store` object called `createStore()`. The `createStore()` helper function has a single argument, a reducer function.

To create a `store` with `lightSwitchReducer`, you could write:

```
import { createStore } from 'redux'

const initialState = 'on';
const lightSwitchReducer = (state = initialState, action) =>
{
  switch (action.type) {
    case 'toggle':
      return state === 'on' ? 'off' : 'on';
    default:
      return state;
  }
}

const store = createStore(lightSwitchReducer);
```

For the remainder of this lesson, you will be building a simple counter application, whose state is a single number, using the Redux library.

In the code editor, you will find the `initialState` value as well as `countReducer`, which describes how the state can be updated in response to an `'increment'` action.

Instructions

1.

First, import the `createStore` method from `redux`.

Checkpoint 2 Passed

Hint

To import the `createStore` method from Redux, you'll write a line that looks like this:

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

2.

Now, below the `countReducer` function, declare a variable called `store`.

Then, call `createStore()` with `countReducer` as the argument. Assign the returned value to `store`.

Checkpoint 3 Passed

Hint

The syntax for using `createStore()` looks like this:

```
const store = createStore(reducerFunction);
```

store.js

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

const initialState = 0;
const countReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'increment':
      return state + 1;
    default:
      return state;
  }
}
```

```
const store = createStore(countReducer);
```

Dispatch Actions to the Store

The `store` object returned by `createStore()` provides a number of useful methods for interacting with its state as well as the reducer function it was created with.

The most commonly used method, `store.dispatch()`, can be used to dispatch an action to the store, indicating that you wish to update the state. Its only argument is an action object, which must have a `type` property describing the desired state change.

```
const action = { type: 'actionDescriptor' };
store.dispatch(action);
```

Each time `store.dispatch()` is called with an `action` object, the store's reducer function will be executed with the same `action` object. Assuming that the `action.type` is recognized by the reducer, the state will be updated and returned.

Let's see how this works in the lightswitch application from the last exercise:

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

const initialState = 'on';
const lightSwitchReducer = (state = initialState, action) =>
{
  switch (action.type) {
    case 'toggle':
      return state === 'on' ? 'off' : 'on';
    default:
      return state;
  }
}

const store = createStore(lightSwitchReducer);

console.log(store.getState()); // Prints 'on'

store.dispatch({ type: 'toggle' });
```

```
console.log(store.getState()); // Prints 'off'

store.dispatch({ type: 'toggle' });
console.log(store.getState()); // Prints 'on'
```

In this example, you can also see another `store` method, `store.getState()`, which returns the current value of the store's state. Printing its value between each dispatched action allows us to see how the store's state changes.

Internally, when the `store` executes its reducer, it uses `store.getState()` as the `state` argument. Though you won't see it, you can imagine that, when an action is dispatched like this…

```
store.dispatch({ type: 'toggle'});
```

…the store calls the reducer like this:

```
lightSwitchReducer(store.getState(), { type: 'toggle' });
```

Instructions

1.

Let's get back to our counter application. The count starts at `0` and we want to increment it up to `2`.

At the bottom of **store.js** dispatch two actions with a type of `'increment'` using the `store.dispatch()` method.

Checkpoint 2 Passed

Hint

To dispatch an action, you can use the following syntax:

```
store.dispatch({ type: 'actionType' })
```

Make sure that the `type` value should match the name of the action in the reducer. In this case, the type of the action should be `'increment'`.

2.

At the bottom of **store.js**, confirm that the current state of the store is `2` by printing out the current value of the state to the console.

Checkpoint 3 Passed

Hint

To get the current state of the store, use `store.getState()`

3.

Let's modify the `countReducer` function so that it can handle `'decrement'` actions as well.

Add an additional case to the `countReducer` function that handles `'decrement'` action types and returns `state - 1`.

Checkpoint 4 Passed

Hint

Your `countReducer` function should look like this:

```
const countReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'increment':
      return state + 1;
    case 'decrement':
      return state - 1;
    default:
      return state;
  }
}
```

4.

Now, let's dispatch some `'decrement'` action to the store.

At the bottom of **store.js** , dispatch 3 actions with a type of `'decrement'`.

Checkpoint 5 Passed

Hint

To dispatch an action of any given type you can write:

```
store.dispatch({ type: 'actionType' });
```

5.

Finally, print to the console the final value of `store.getState()`. The final state should be `-1`.

store.js

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

// Create your action creators here.
```

```
const initialState = 0;
const countReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'increment':
      return state + 1;
    case 'decrement':
      return state - 1;
    default:
      return state;
  }
}

const store = createStore(countReducer);

// Modify the dispatches below.
store.dispatch({ type: 'increment' });
store.dispatch({ type: 'increment' });
console.log(store.getState());

store.dispatch({ type: 'decrement' });
store.dispatch({ type: 'decrement' });
store.dispatch({ type: 'decrement' });
console.log(store.getState());
```

Action Creators

As you saw in the last exercise, you are likely to dispatch actions of the same type multiple times or from multiple places. Typing out the entire action object can be tedious and creates opportunities to make an error.

For example, in the light switch application, whose reducer accepts `'toggle'` actions to turn the light `'on'` or `'off'`, you might write:

```
store.dispatch({Type:'toggle'});
store.dispatch({type:'toggel'});
store.dispatch({typo:'toggle'});
```

Did you spot the errors?

In most Redux applications, *action creators* are used to reduce this repetition and to provide consistency. An action creator is simply a function that returns an action object with a `type` property. They are typically called and passed directly to the `store.dispatch()` method resulting in fewer errors and an easier-to-read dispatch statement.

The above code could be rewritten using an action creator called `toggle()` like so:

```
const toggle = () => {
  return { type: "toggle" };
}
store.dispatch(toggle()); // Toggles the light to 'off'
store.dispatch(toggle()); // Toggles the light back to 'on'
store.dispatch(toggle()); // Toggles the light back to 'off'
```

Though not necessary in a Redux application, action creators save us the time needed to type out the entire action object, reduce the chances you make a typo, and improve the readability of our application.

Often, before the reducer of an application is even written, Redux programmers will write action creators as a way of planning out which actions will be available to dispatch to the store.

Instructions

1.

Let's add some action creators into the counter application. The two actions that the `countReducer` can handle are `'increment'` and `'decrement'`.

First, at the top of the **store.js** file, create an action creator function called `increment()` that returns an object with a `type: 'increment'` property.

Checkpoint 2 Passed

Hint

A generic action creator function may look like this:

```
const actionType = () => {
  return { type: 'actionType' };
}
```

2.

Well done! Now, after the `increment()` action creator and before the `countReducer`, add in a second action creator named `decrement()` that returns an action object with the `'decrement'` action type.

Checkpoint 3 Passed

Hint

If `increment()` action creator looks like this…

```
const increment = () => {
  return { type: 'increment' };
}
```

…then what would the `decrement()` action creator look like?

3.

In **store.js**, actions are being dispatched to the store like so:

```
store.dispatch({ type: 'increment' });
store.dispatch({ type: 'decrement' });
```

For each existing call to `store.dispatch()`, replace the typed-out action objects with function calls to the appropriate action creator.

Checkpoint 4 Passed

Hint

Action creators can be passed directly to the `store.dispatch()` method as an argument:

```
store.dispatch(actionCreator());
```

Make sure to replace all 5 `store.dispatch()` arguments!

store.js

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

// Create your action creators here.
const increment = () => {
  return { type: 'increment'};
}

const decrement = () => {
```

```
  return { type: 'decrement'};
}

const initialState = 0;
const countReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'increment':
      return state + 1;
    case 'decrement':
      return state - 1;
    default:
      return state;
  }
}

const store = createStore(countReducer);

// Modify the dispatches below.
store.dispatch(increment());
store.dispatch(increment());
console.log(store.getState());

store.dispatch(decrement());
store.dispatch(decrement());
store.dispatch(decrement());
console.log(store.getState());
```

Respond to State Changes

In a typical web application, user interactions that trigger DOM events ("click", "keydown", etc…) can be listened for and responded to using an event listener.

Similarly, in Redux, actions dispatched to the `store` can be listened for and responded to using the `store.subscribe()` method. This method accepts one argument: a function, often called a *listener*, that is executed in response to changes to the `store`'s state.

```
const reactToChange = () => console.log('change detected!');
store.subscribe(reactToChange);
```

In this example, each time an action is dispatched to the `store`, and a change to the state occurs, the subscribed listener, `reactToChange()`, will be executed.

Sometimes it is useful to stop the listener from responding to changes to the `store`, so `store.subscribe()` returns an `unsubscribe` function.

We can see this in action in the light switch application:

```
// lightSwitchReducer(), toggle(), and store omitted...

const reactToChange = () => {
  console.log(`The light was switched
${store.getState()}!`);
}
const unsubscribe = store.subscribe(reactToChange);

store.dispatch(toggle());
// reactToChange() is called, printing:
// 'The light was switched off!'

store.dispatch(toggle());
// reactToChange() is called, printing:
// 'The light was switched on!'

unsubscribe();
// reactToChange() is now unsubscribed

store.dispatch(toggle());
// no print statement!

console.log(store.getState()); // Prints 'off'
```

- In this example, the listener function `reactToChange()` is subscribed to the `store`
- Each time an action is dispatched, `reactToChange()` is called and prints the current value of the light switch. It is common for callbacks subscribed to the `store` to use `store.getState()` inside them.
- After the first two dispatched actions, `unsubscribe()` is called causing `reactToChange()` to no longer be exectued in response to further dispatches made to `store`.

Note: It is not always required to use the `unsubscribe()` function returned by `store.subscribe()`, though it is useful to know that it exists.

Now, take a look at **store.js** in the code editor. You will see that a few actions have been dispatched to the `store` of the counter application. Suppose you wanted to print the current value of `store.getState()` each time the state changes. While you could write something like this…

```
store.dispatch(decrement());
console.log(`The count is ${store.getState()}`);
store.dispatch(increment());
console.log(`The count is ${store.getState()}`);
store.dispatch(increment());
console.log(`The count is ${store.getState()}`);
```

…we know that this approach is repetitive. Instead, you can subscribe a change listener to print out the current state in response to state changes automatically.

Instructions

1.

The first thing to do is to define a state change listener.

Define a function called `printCountStatus()` with no arguments. It should print to the console the following message:

```
console.log(`The count is ${REPLACE_WITH_CURRENT_STATE}`);
```

Make sure to replace `REPLACE_WITH_CURRENT_STATE` with the proper code for getting the current state from the `store`.

Checkpoint 2 Passed

Hint

Your `printCountStatus()` function should look like this:

```
const printCountStatus = () => {
  console.log(`The count is ${store.getState()}`);
}
```

2.

Now that you have a change listener function, subscribe it to the `store` so that it is called each time the state changes.

If done correctly, you should see this in the console output:

```
The count is -1
The count is 0
The count is 1
Checkpoint 3 Passed
```

Hint

To subscribe a change listener to the `store`, you can write:

Make sure not to call the `listener` when you pass it to `store.subscribe()`!

```
store.subscribe(listener);
```

store.js

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

const increment = () => {
  return { type: 'increment' }
}

const decrement = () => {
  return { type: 'decrement' }
}

const initialState = 0;
const countReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'increment':
      return state + 1;
    case 'decrement':
      return state - 1;
    default:
      return state;
  }
}

const store = createStore(countReducer);
```

```
// Define your change listener function here.
function printCountStatus() {console.log(`The count is ${sto
re.getState()}`)};
store.subscribe(printCountStatus);

store.dispatch(decrement()); // store.getState() === -1
store.dispatch(increment()); // store.getState() === 0
store.dispatch(increment()); // store.getState() === 1
```

```
The count is -1
The count is 0
The count is 1
```

Connect the Redux Store to a UI

Up until now, you have built a working counting application using Redux that lacks a proper user interface (UI). Let's change that!

The UI for this application should display the current count number and allow the user to increment or decrement this value using the buttons provided. Take a look at the connected web browser window and you can see that the elements for such an interface are present, but they haven't been connected to the Redux store yet.

Connecting a Redux store with any UI requires a few consistent steps, regardless of how the UI is implemented:

- Create a Redux store
- Render the initial state of the application.
- Subscribe to updates. Inside the subscription callback:
    - Get the current store state
    - Select the data needed by this piece of UI
    - Update the UI with the data
- Respond to UI events by dispatching Redux actions

These same steps are followed when building an interface using React, Angular, or jQuery. For now, we'll create a very simple user interface for the counting application using the HTML DOM API.

Open up the **index.html** file and you can see the three HTML elements that are currently being rendered:

```
<p id='counter'>Waiting for current state.</p>
<button id='incrementer'>+</button>
<button id='decrementer'>-</button>
```

Now, open up **store.js** where you will find the pieces of Redux code that you have built throughout this lesson: the action creators `increment()` and `decrement()`, the reducer `countReducer`, and the `store` that ties it all together. Additionally, the following values have been added:

- `counterElement`, `incrementer`, and `decrementer`: references to the HTML elements in **index.html**
- `render`: A state-change listener for responding to changes to the `store`'s state.
- `incrementerClicked` and `decrementerClicked`: DOM event handlers for responding to the buttons being clicked by the user.

These new functions and elements will allow us to plug the Redux `store` into the UI. Let's begin.

Instructions

1.

The `counterElement` should display the current value of the store, but currently it is displaying the message "Waiting for current state." You can change this text by assigning a new value to `counterElement.innerHTML` like so:

```
counterElement.innerHTML = "New text to display";
```

Within the `render()` function, reassign `counterElement.innerHTML` to display the current state of the `store` in the UI.

Then, after the definition of `render()`, call it once to render the initial state of the `store`.

Checkpoint 2 Passed

Hint

To get the current state of the `store`, use the `store.getState()` method. Your code should look like this:

```
const render = () => {
  counterElement.innerHTML = the_current_state_of_the_store
};
render();
```

When you're finished, the `counterElement` should be showing `0`, the initial state of the `store`.

2.

The `incrementerClicked()` function will be called each time the `incrementer` button is clicked by the user. When this happens, the `store` should be notified and the state should be incremented by `1`.

Inside `incrementerClicked()`, use `store.dispatch()` and and the appropriate action creator to tell the `store` to increase its state by 1.

Note: completing this checkpoint will not cause the UI to change (you'll see why soon).

Checkpoint 3 Passed

Hint

To dispatch an action to the store using `store.dispatch()` and an `actionCreator()` function, write:

```
store.dispatch(actionCreator());
```

You'll want to use the `increment()` action creator for this one!

3.

Now, pressing the `incrementer` button will send a `{ type: 'increment' }` action object to the `store` which increases the value of the state to `1`. But, the UI doesn't seem to be updating.

In order for the UI to react to changes to the state of the `store`, you have to subscribe a change listener to the `store` using `store.subscribe()`!

Below the declaration of the `render()` function, call `store.subscribe()` and pass in `render` as the argument so

that the UI re-renders each time the state of the `store` changes. Then, try clicking on the “+” button.

Checkpoint 4 Passed

Hint

To subscribe a change listener, you can write:

```
store.subscribe(listener);
```

4.

Nicely done! Press the `incrementer` button and you should see the counter increase in the UI! All that’s left to do is get the `decrementer` button’s event handler to work.

Within `decrementerClicked()`, dispatch a `decrement()` action to the `store`.

Checkpoint 5 Passed

Hint

To dispatch a `decrement()` action to the `store`, you can write:

```
store.dispatch(decrement());
```

store.js

```
/* Note to learners:
Normally, you would import redux like this:

  import { createStore } from 'redux';

Due to Codecademy's technical limitations
for testing this exercise, we are using
`require()`.
*/
const { createStore } = require('redux');

// Action Creators
function increment() {
  return {type: 'increment'}
}

function decrement() {
```

```
  return {type: 'decrement'}
}

// Reducer / Store
const initialState = 0;
const countReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'increment':
      return state + 1;
    case 'decrement':
      return state - 1;
    default:
      return state;
  }
};
const store = createStore(countReducer);

// HTML Elements
const counterElement = document.getElementById('counter');
const incrementer = document.getElementById('incrementer');
const decrementer = document.getElementById('decrementer');

// Store State Change Listener
const render = () => {
  counterElement.innerHTML = store.getState();
}

store.subscribe(render);

render();


// DOM Event Handlers
const incrementerClicked = () => {
  store.dispatch(increment());
}
incrementer.addEventListener('click', incrementerClicked);

const decrementerClicked = () => {
  store.dispatch(decrement());
```

```
}
decrementer.addEventListener('click', decrementerClicked);
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="./index.css">
  <title>Learn ReactJS</title>
</head>

<body>
  <main id="app">
    <p id='counter'>Waiting for current state.</p>
    <button id='incrementer'>+</button>
    <button id='decrementer'>-</button>
  </main>

</body>
<!-- Do Not Remove -->
<script src="https://content.codecademy.com/courses/React/react-16-redux-4-bundle.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/redux/4.0.5/redux.min.js" integrity="sha512-P36ourTueX/PrXrD4Auc1kVLoTE7bkWrIrkaM0IG2X3Fd90LFgTRogpZzNBssay0XOXhrIgudf4wFeftdsPDiQ==" crossorigin="anonymous"></script>
<script src="./store.js"></script>
</html>
```

React and Redux

As you saw in the last exercise, Redux can be used within the context of any UI framework, though it is most commonly

paired with React. This makes sense considering that React and Redux were both developed by engineers at Facebook.

We can be more specific about the common steps involved in connecting Redux to a React UI:

- A `render()` function will be subscribed to the `store` to re-render the top-level React Component.
- The top-level React component will receive the current value of `store.getState()` as a `prop` and use that data to render the UI.
- Event listeners attached to React components will dispatch actions to the `store`.

Take a look at **store.js** in the code editor. Here, you can see the completed light switch application following this pattern.

- The `render()` function is subscribed to the `store`.
- `store.getState()` is passed as a `prop` called `state` to the `<LightSwitch />` component.
- The `LightSwitch` component displays the current state of the store, either `'on'` or `'off'`, and adjusts the background colors accordingly.
- The `LightSwitch` component declares a click handler that dispatches a `toggle()` action to the `store`.

Note 1: The prop name `state` isn't a special React name and can be customized as the programmer sees fit. For example, `lightSwitchState={store.getState()}` would also be valid.

Instructions

In the next exercise, you will implement the Counter app using a React UI. For now, take a moment and familiarize yourself with how the state of the `store` is used by this application's React components.

Observe that the same steps from the last exercise for connecting Redux to a UI are followed:

- Create a Redux store
- Render the initial state of the application.
- Subscribe to updates. Inside the subscription callback:
    - Get the current store state

    - Select the data needed by this piece of UI
    - Update the UI with the data
- Respond to UI events by dispatching Redux actions

store.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore } from 'redux';

// REDUX CODE
/////////////////////////////////////

const toggle = () => {
  return {type: 'toggle'}
}

const initialState = 'off';
const lightSwitchReducer = (state = initialState, action) =>
 {
  switch (action.type) {
    case 'toggle':
      return state === 'on' ? 'off' : 'on';
    default:
      return state;
  }
}

const store = createStore(lightSwitchReducer);

// REACT CODE
/////////////////////////////////////

// Pass the store's current state as a prop to the LightSwit
ch component.
const render = () => {
  ReactDOM.render(
    <LightSwitch
      state={store.getState()}
    />,
    document.getElementById('root')
```

```
  )
}

render(); // Execute once to render with the initial state.
store.subscribe(render); // Re-
render in response to state changes.

// Receive the store's state as a prop.
function LightSwitch(props) {
  const state = props.state;

  // Adjust the UI based on the store's current state.
  const bgColor = state === 'on' ? 'white' : 'black';
  const textColor = state === 'on' ? 'black' : 'white';

  // The click handler dispatches an action to the store.
  const handleLightSwitchClick = () => {
    store.dispatch(toggle());
  }

  return (
    <div style={{background : bgColor, color: textColor}}>
      <button onClick={handleLightSwitchClick}>
        {state}
      </button>
    </div>
  )
}
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="./index.css">
  <title>Learn ReactJS</title>
</head>

<body>
```

```
  <div id="root">
  </div>
</body>
<!-- Do Not Remove -->
<script src="https://content.codecademy.com/courses/React/react-16-redux-4-bundle.min.js"></script>
<script src="./store.compiled.js"></script>
</html>
```

Implementing a React+Redux App

Now that you have implemented the counter app using the HTML DOM API, and have seen a working React+Redux application, it is time to implement it using React.

Take a look at the **store.js** file and you will find the following functions and values have been defined for you:

- The action creators `increment()` and `decrement()`
- The `store` and its reducer `countReducer()`
- A React component called `CounterApp` which declares two event handlers, `onIncrementButtonClicked` and `onDecrementButtonClicked`
- A `render()` function which renders `CounterApp` using `ReactDOM.render()`

The React component `CounterApp` and the `render()` function are entirely disconnected from the Redux store. Let's change that!

Instructions

1.
At this point, you should be familiar with the desired functionality of this UI. Notice that nothing is currently being rendered in the web browser.

First, below the `render()` function's definition, call `render()` once to render the UI with the initial state.

Checkpoint 2 Passed

Hint

Make sure to call `render()` *after* its definition. Your code should look like this:

```
const render = () => {
  ReactDOM.render(
    <CounterApp

    />,
    document.getElementById('root')
  )
}
render();
```

2.

The `CounterApp` component should display the current state of the `store`. Before it can display the current state, it must be given the current state value.

Within the `render()` function, pass a prop value to `CounterApp` called `state`. Its value should be the current state of the store. Your `render()` function should look something like this:

```
const render = () => {
  ReactDOM.render(
    <CounterApp
      state={currentStateValueGoesHere}
    />,
    document.getElementById('root')
  )
}
```

Checkpoint 3 Passed

Hint

Use `store.getState()` to get the current state of the store.

3.

Now that the current state of the `store` is being passed to the `CounterApp` component, it can render that data in the UI.

First, at the top of the `CounterApp()` function, declare a variable called `state`. It should be assigned the value of `props.state`.

Then, modify the `<h1>` element inside the `return` statement of `render()` such that it displays the current `state`.

Checkpoint 4 Passed

Hint

Hint: At this point, the `return` statemenet of your `CounterApp` should look like this:

```
return (
  <div>
    <h1> {the_state_from_props} </h1>
    <button onClick={onIncrementButtonClicked}>+</button>
    <button onClick={onDecrementButtonClicked}>-</button>
  </div>
)
```

4.

At this point, your user interface should be displaying the current state of the store, `0`. The next step is to update the state when either of the buttons are pressed.

First, modify the `onIncrementButtonClicked` event handler such that it dispatches an `increment()` action to the `store`.

Then, modify the `onDecrementButtonClicked` event handler such that it dispatches a `decrement()` action to the `store`.

Checkpoint 5 Passed

Hint

To dispatch an action from an action creator to the store, you can write:

```
store.dispatch(actionCreator());
```

5.

Well done! So far we can display the current state and dispatch action from the `<CounterApp />` - all that's left is to re-render the component every time the state changes.

At the bottom of **store.js**, subscribe the `render` function to the `store`.

Checkpoint 6 Passed

Hint

You can subscribe a listener to respond to changes to the `store`'s state like so:

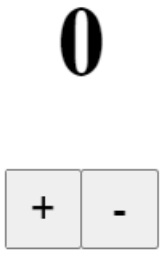
```
store.subscribe(listener);
```

store.js

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

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

// REDUX CODE
//////////////////////////////////

const increment = () => {
  return {type: 'increment'}
}

const decrement = () => {
  return {type: 'decrement'}
}

const initialState = 0;
const counterReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'increment':
      return state + 1;
    case 'decrement':
      return state - 1;
    default:
      return state;
  }
}

const store = createStore(counterReducer);

// REACT CODE
//////////////////////////////////

const render = () => {
  ReactDOM.render(
    <CounterApp
      state={store.getState()}
    />,
    document.getElementById('root')
  )
}

render();
```

```
store.subscribe(render);


// Render once with the initial state.
// Subscribe render to changes to the store's state.

function CounterApp(props) {
  const state = props.state;

  const onIncrementButtonClicked = () => {
    // Dispatch an 'increment' action.
    store.dispatch(increment());
  }

  const onDecrementButtonClicked = () => {
    // Dispatch an 'decrement' action.
    store.dispatch(decrement());
  }

  return (
    <div id='counter-app'>
      <h1> {state} </h1>
      <button onClick={onIncrementButtonClicked}>+</button>
      <button onClick={onDecrementButtonClicked}>-</button>
    </div>
  )
}
```



Review

Congratulations! You were able to apply the core concepts of the Redux framework by implementing an application using the Redux library.

By completing this lesson, you are now able to:

- Install the redux library into your project using `npm install redux`.
- Import the `createStore()` helper function from the `'redux'` library.
- Create a `store` object that holds the entire state of your Redux application using `createStore()`.
- Get the current state of the `store` using `store.getState()`.
- Dispatch actions to the `store` using `store.dispatch(action)`.
- Create action creators to reduce the repetitive creation of action objects.
- Register a change listener function to respond to changes to the store using `store.subscribe(listener)`.
- Recognize the pattern for connecting Redux to any user interface.
- Implement a Redux application using either the HTML DOM API or React.

Throughout this lesson, you may have thought to yourself that Redux adds a lot of unnecessary complexity to these simple applications. We implemented Redux in a very basic way, which is useful for learning but not how it’s done in the real world.

Redux shines when it is used in applications with many features and a lot of data where having a centralized store to keep it all organized is advantageous. In the next lesson, you will learn how to build and organize Redux applications with complex state.