Learn Redux

Codecademy

Intro to the Core Redux API

What is the Redux API?

Install the Redux Library

The core concepts of **Redux** are closely tied to a framework known as **Flux**. Both share the same concept of 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 Node Package Manager (npm). Then, its methods can be imported.

Run **npm install redux** in bash terminal.

In the Store.js **import { createStore } from 'redux';**

Create a Redux 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.

```
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);
```

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 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.

```
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'
```

`store.getState()`, returns the current value of the store's state.

Internally, when the store executes its reducer, it uses `store.getState()` as the state argument.

Action Creators

Typing out entire action object can be tedious and creates opportunities to make error.

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.

```
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 for typo, and improve the readability of our application.

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.

```
// 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 executed in response to further dispatches made to `store`.

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

Connect the Redux Store to a UI

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 of 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.

React and Redux

Redux can be used within the context of any UI framework, though it is most commonly paired with React as they 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`

```
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 LightSwitch 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>
  )
}
```

Implementing a React+Redux App

```
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();
const subscribe = 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>
```

```
  )
}
```