Listed App with Redux

Listed With Redux

Listed With Redux

Let's revisit our app from yesterday, and rebuild it using Redux. Again the disclaimer needs to be said that using Redux for an application as simple as this is probably overkill, but we'll be able to replicate the implimentation of Redux in a more realistic production-like setup.

Since Redux requires so much boilerplate code, we've created a starter repo here.

React/Redux Starter Repo

```
mkdir listed-redux
```

```
cd listed-redux

git clone https://github.com/HackerYou/react-redux-starter .

npm install

npm start
```

This boilerplate has folders for both `reducers` and `actions` and uses imports to include them where they are needed.

Adding styles

Copy and paste the styles from this file into your `index.css`.

Listed CSS

We'll also need to include an images folder, which you can unzip here and move into the `src` folder:

Images

The Reducer & combineReducers

Let's start in our `reducers/index.js`. We have a bit of boilerplate code here that will allow our app to run initially. This boilerplate introduces a new method, `combineReducers`, which for now returns an empty state object but in the future will allow for us to further componentize our reducers - in a complex application we might have a large number of pieces of state to manage, and it could get unwieldy if it is all done in one file.

In our listed application, one piece of the application state is the lists - but the future, we might add a piece of state for users, and whatever else, too.

Let's break out our list reducer into it's own file. Create a new file in the `reducers` folder called `reducer_lists.js`. It doesn't matter what this file is called but this is a nice convention.

Inside of our new file, add this code:

```
export default function(state = {}, action) {
```

```
  return state;
}
```

And now in `reducers/index.js` we need to import this piece of state and pass it to our `combineReducers` method:

```
import { combineReducers } from 'redux';
import ListReducer  from './reducer_lists';

const rootReducer = combineReducers({
  lists: ListReducer
});


export default rootReducer;
```

The `combineReducers` function is going to take all of our pieces of state and "reduce" them to one large object - a single piece of state.

Adding Components

Let's get that code we need to build our app back up and running. Nagivate to `components/App.js` and replace the code with the following. There are commented out bits that we will bring back in once things are connected properly!

```
import React, { Component } from "react";
import '../index.css';

class App extends Component {

  render() {
    return (
      <div className="App">
        <header>
          <h1>Listed</h1>
```

```
            <form>
              <input
                type="text"
                placeholder="add a list"
              />
            </form>
          </header>
          <main>Lists go here</main>
        </div>
      );
    }
  }


export default App;
```

Similar to in yesterday's example, probably the first thing that we want to do is...get the lists. We'll still be using the `componentDidMount` lifecycle hook to drigger this, but instead of doing our request right here and adding it ot application state, we're going to trigger an action on mount of the component in order to pass the lists on to our reducer and add it to our Redux store.

We know now that a Redux `action` is a JavaScript object which requires a `type` property. What we want to do is make an asynchronous request to an API, and when that data comes back, take the result of the request and add it into state. This kind of complicates things, because what we need to pass along is a `Promise`.

Redux Middleware

What is middleware? Functions taht take an action and deending on thea ction's type or payload, the middleware can decide to let the action pass through, manipulate it, or stop it before they hit the reducers. They act like a gatekeeper that stops any action and performs some sort of task before they are let through.

We can have unlimited amounts of middleware in our application depending on what our needs are. In our case, we're going to be using a package called `redux-promise` which will allow us to nicely work with async requests.

In your terminals, run:

```
npm install --save redux-promise
```

Now we need to wire it up to our application. Start up the project again with `npm start` and navigate to `src/index.js`.

We have already set up this boilerplate to handle middleware with the `applyMiddleware` method from Redux (another higher order function!) Let's edit this file to import `redux-promise` and apply it here.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { createStore, applyMiddleware } from 'redux';
import promise from 'redux-promise';

import App from './components/App';
import reducers from './reducers';

const createStoreWithMiddleware = applyMiddleware(promise)(createStore)

ReactDOM.render(
  <Provider
    store={createStoreWithMiddleware(
      reducers,
      window.__REDUX_DEVTOOLS_EXTENSION__ &&
        window.__REDUX_DEVTOOLS_EXTENSION__()
    )}
  >
    <App />
  </Provider>,
  document.getElementById("root")
);
```

That's all we need to do to set it up. Now, `redux-promise` is going to allow us to make our AJAX

request and pass along the promise as the payload of our action rather than plain data.

Action Creators

In our simple counter example, we directly passed an action into the `dispatch` method provided to us by Redux. Sometimes if you have some business to perform in your action that is more complex than just a simple `increment`, it is more idiomatic to use an *action creator* (A function that returns an action). We're goign to do this here.

Navigate to `actions/index.js` and let's add this code:

```
import axios from "axios";

export const FETCH_LISTS = "fetch_lists";

// saves the root url as a variable
const rootUrl = `http://lists.hackeryou.com`;

export function fetchLists() {
  // sets up an axios request that returns a promise
  const request = axios.get(`${rootUrl}/list`);

  return {
    type: FETCH_LISTS,
    // the promise itself is the payload, which is why we need to use t
    payload: request
  };
}
```

Action creators need to return an action, and an action always needs to have a `type`. In our last example, we put a string of `'INCREMENT_COUNT'`. Here, we made it into a separate variable, made it a named export, and assigned it to the string `"fetch_lists"`. The capital letters are used to indicate a constant.

Using a variable helps to keep our action types consistant between action creators and reducers (less chance of typos!).

Then, we make our request using the `axios` library and pass the promise into the `payload`. (Note that unlike `type`, we can call the `payload` property anything we want - this is just a convention also.)

Adding Action Creators to Components

Our goal is to call this action creator that gets the lists for us whenever the app loads. Let's head back to `components/App.js` and write that code.

First thing we need to do is wire up our `App` component to Redux using the `connect` method. We also need to bind our action creator `fetch_lists` as a prop on the component, in a similar way that we were mapping state to props in the previous example.

The `mapDispatchToProps` method we are defining below the component returns a call to `bindActionCreators`. This causes the action creator to be bound to dispatch and flows through our middleware before heading to our reducers.

Finally, we are exporting `connect` with `null` as the first argument and `mapDispatchToProps` as the second. We have to pass `null` in first, because the first argument expects the `mapStateToProps` method that we are not using here.

```
import React, { Component } from "react";
import { connect } from 'react-redux';
import { bindActionCreators } from 'redux';
import { fetchLists } from '../actions/index';


import '../index.css';

class App extends Component {

  render() {
    return (
      <div className="App">
        <header>
          <h1>Listed</h1>
          <form>
            <input
```

```
              type="text"
              placeholder="add a list"
            />
          </form>
        </header>
        <main>Lists go here</main>
      </div>
    );
  }
}

function mapDispatchToProps(dispatch){
  return bindActionCreators({ fetchLists }, dispatch)
}
export default connect(null, mapDispatchToProps)(App);
```

Calling The Action Creator

We want to actually call our action creator when the component mounts - so let's add that code to our `App` component now. For the moment, let's just `console.log` the result of this.

```
componentDidMount(){
    this.props.fetchLists()
      .then(res => console.log(res));
  }
```

You should see something like this come back in the console:

```
{type: 'fetch_lists', payload: {...}}
```

That's great! We've successfuly added all of our action creators to the app, and called the method when the component mounted, which is then triggering an ajax request and passing along the returned data to the `payload` property of our action.

Deeper Look at Redux-Promise

Let's now add a reducer that will handle the part of state where we store our lists. Hop over to `reducer_lists.js`.

For now to get an idea of what's happening when, add this to the file:

```
//reducer_lists.js
export default function(state = {}, action) {
  console.log('Action!!', action)
  return state;
}
```

Remember here that we are exporting this reducer and importing it to `reducers/index.js` using the `combineReducers` method to add this part of state to the Redux store.

Let's take a look in the console at what's happening now.

Notice that Redux does eventually dispatch our action, `fetch_lists`, but before that it dispatches a few actions of it's own. This is why it's important that we always have a default case written in our reducers to return state.

Let's also add a console log to `actions/index.js`:

```
//actions/index.js
import axios from "axios";

export const FETCH_LISTS = "fetch_lists";

// saves the root url as a variable
const rootUrl = `http://lists.hackeryou.com`;

export function fetchLists() {
  // sets up an axios request that returns a promise
  const request = axios.get(`${rootUrl}/list`);

  console.log('request:', request)
  return {
    type: FETCH_LISTS,
```

```
    // the promise itself is the payload, which is why we need to use t
    payload: request
  };
}
```

The important thing to note here is that the request is what we are passing in as the payload.

If you look at the request, you can see that it is a `promise`.

The action typically goes straight to the reducers. It's `redux-promise` that has the ability to stop and resolve the promise before it hits the reducer. This makes things a lot nicer to work with, because we have the data available to us immediately rather than having to work with the promise ourselves.

Updating Our Reducer

Let's add the switch statement that will inform the reducer how we want it to repond based on various actions occuring.

We need to import our constant variable that we have as a reference to our action in the `actions` file. We're also like yesterday going to be using the underscore library's `_.indexBy` method to create a keyed object rather than an array for ease of updating.

```
import { FETCH_LISTS } from "../actions";

export default function(state = {}, action) {
  switch (action.type) {
    case FETCH_LISTS:
      // this underscores method takes a datalist and a desired key fo
      return _.indexBy(action.payload.data, "_id");
    default:
      return state;
  }
}
```

Since we're getting the entire list of lists with this call, we're done! We don't really have to worry about

immutability here (we will soon!).

Mapping State To Props

Time to head back to our `App` component where we want to grab the lists out of the Redux store and pass them along to a method to render them out.

```
//App.js

// Code for component goes here...


function mapStateToProps(state){
  return {
    lists: state.lists
  }
}

function mapDispatchToProps(dispatch){
  return bindActionCreators({ fetchLists }, dispatch)
}
export default connect(mapStateToProps, mapDispatchToProps)(App);
```

Hot tip: we could use ES6 to refactor our `mapStateToProps` method to be this:

```
function mapStateToProps({ lists }) {
  return {
    lists
  }
}
```

Pretty nice, right!?

Now, instead of our React application state being the place our Lists are stored, it's available to us from the Redux store under the `lists` property - and we've added it as a `prop` on our `App`

component. Let's go ahead and render those out, and add our `List` and `ListItem` components while we are at it.

`App.js`

```
//App.js
import React, { Component } from "react";
import { connect } from 'react-redux';
import { bindActionCreators } from 'redux';
import { fetchLists } from '../actions/index';
import List from './List';


import '../index.css';

class App extends Component {
  constructor(){
    super();
    this.renderLists = this.renderLists.bind(this);
  }
  componentDidMount(){
    this.props.fetchLists();
  }
  renderLists(){
    const lists = this.props.lists;
    const listKeys = Object.keys(this.props.lists);

    return listKeys.map((listKey) => {
      return <List key={listKey} listInfo={lists[listKey]}/>
    });
  }
  render() {
    return <div className="App">
        <header>
          <h1>Listed</h1>
          <form>
            <input type="text" placeholder="add a list" />
          </form>
        </header>
```

```
        <main> {this.renderLists()}</main>
      </div>;
  }
}

function mapStateToProps(state){
  return {
    lists: state.lists
  }
}

function mapDispatchToProps(dispatch){
  return bindActionCreators({ fetchLists }, dispatch)
}
export default connect(mapStateToProps, mapDispatchToProps)(App);
```

List.js

```
//List.js
import React, { Component } from "react";

import ListItem from "./ListItem";

// images
import latest from "../images/latest.png";
import popular from "../images/popular.png";

class List extends Component {
  constructor() {
    super();
    this.state = {
      sortBy: "latest"
    };
    this.sortItems = this.sortItems.bind(this);
    this.setSortBy = this.setSortBy.bind(this);
  }
  setSortBy(e) {
    this.setState({
      sortBy: e.target.id
```

```
    });
  }
  sortItems() {
    const popular = (a, b) => b.score - a.score;
    const latest = (a, b) => b.created_at - a.created_at;

    const sortedList = Array.from(this.props.listInfo.items).sort(
      this.state.sortBy === "popular" ? popular : latest
    );

    const listItems = sortedList.map(item => {
      return (
        <ListItem
          key={item._id}
          listId={this.props.listInfo._id}
          itemInfo={item}
        />
      );
    });

    return listItems;
  }
  render() {
    return (
      <div className="list">
        <header>
          <h2>{this.props.listInfo.title}</h2>
          <div className="list__sorting">
            <p>Sort by:</p>

            <button id="popular" onClick={this.setSortBy}>
              <img src={popular} alt="thumbs up icon" />Most Popular
            </button>
            <button id="latest" onClick={this.setSortBy}>
              <img src={latest} alt="" />Latest
            </button>
          </div>
        </header>
        <ul>
          {this.sortItems()}
```

```
            <li className="list__item">
              <form>
                <label htmlFor="newItem" className="visually-hidden">
                  Add To List:
                </label>
                <input type="text" id="newItem" placeholder="add item to
              </form>
            </li>
          </ul>
        </div>
      );
    }
  }


export default List;
```

Something to note in our `List` component is that we've already included the code to sort our items by popularity or most recent. This is exactly the same code we used yesterday - and we *are* using the Component state to do it. You'll find different opinions on when to use the Redux store vs. local state, and typically we follow the *WWDD* (What Would Dan Do) approach to Redux - if it doesn't matter to the app globally and doesn't mutate in complex ways - such as some UI toggle, a form input, or in this case - the order in which items from the store appear - React state is just fine.

Thanks, Dan!

`ListItem.js`

```
//ListItem.js
import React from "react";

// images
import upvote from '../images/upvote.png';
import downvote from '../images/downvote.png';

const ListItem = ({ itemInfo }) => {
  return (
    <li className="list__item">
```

```
        {itemInfo.item}
        <div className="item__scoreCard">
          <span className="scoreCard__score">{itemInfo.score}</span>
          <button>
            <img src={upvote} alt="thumbs up" />
            Upvote
          </button>
          <button>
            <img src={downvote} alt="thumbs down" />Downvote
          </button>
        </div>
      </li>
    );
};

export default ListItem;
```

Adding Items To A List

Now, let's add the ability to add a list item. This will be an interesting challenge because now we want to update / change the state in the Redux store when this action fires - so we'll need to explore how to do this in an immutable way.

We already have the input there to allow the user to add an item to a list. In yesterday's app, we created a *controlled input* to connect the input's inherent state (someone typing some stuff into it) with React's state. We're going to do the same thing here - this is one of those 'ephemeral' state situations where all our big store is concerned about is what the final item is, rather than any key presses.

Let's grab this handleChange method, bind it in the constructor, and add `onChange={this.handleChange} value={this.state.newItem}` to the `input`.

```
  handleChange(e) {
    let newItem = e.target.value;
    this.setState({
      newItem
    });
  }
```

When it comes time to submit the form, it's Redux's time to work again. We'll want to write a new action creator that will add the new item to the api. Let's start from the component this time, and work our way back up.

On the form, add `onSubmit={this.handleSubmit}`.

Our `handleSubmit` method will look like this:

```
handleSubmit(e){
  e.preventDefault();

  let newItem = this.state.newItem.trim();

  if (newItem !== ''){
    this.props.addItem(this.props.listInfo._id, newItem)
  }

  this.setState({
    newItem: ''
  })
}
```

Above, we are calling an *action creator* that we have yet to define - so your app won't compile for the moment.

Let's write our action creator now in `actions/index.js`.

```
//actions/index.js

// ....

export const ADD_ITEM = 'add_item';

// ...

export function addItem(listId, item){
```

```
  const request = axios.post(`${rootUrl}/list/${listId}/item`, { item
  return {
    type: ADD_ITEM,
    payload: request
  }
}
```

Remember here that we are creating a constant for the action type to avoid typos, and we are once again providing our action with a `promise` rather than data. `redux-promise` - our middleware - will resolve this promise and pass the data to our reducers for us.

Back in `List.js`, let's connect this component to Redux and import our action creator:

```
// ....

import { connect } from 'react-redux';
import { bindActionCreators } from 'redux';
import { addItem } from '../actions';
// ....


function mapDispatchToProps(dispatch) {
  return bindActionCreators({ addItem }, dispatch);
}

export default connect(null, mapDispatchToProps)(List);
```

Now, when we add an item to a list - we should see a successfull post request to the API in the network panel. We still need to refresh the page to see any changes, because we haven't yet told the reducer how to handle this action firing.

Let's go back to `reducers/reducer_lists.js` to add a case for an updated list item. Here, we need to ensure that we are not mutating the state - we are making a deep copy of the state and returning a completely new state.

In yesterday's codealong, we used a little hack with `JSON.stringify` and `JSON.parse` which

worked well for our purposes. You can also use a package called immutability-helper to do something similar. Just for *fun*, today we'll use the ES6 spread operator to update our lists. It will look something like this:

```
case ADD_ITEM: {

      const newItem = action.payload.data.item;
      const listId = action.payload.data.item.belongs_to;

      return {
        ...state,
        [listId]: {
          ...state[listId],
          items: [...state[listId].items, newItem]
        }
        }

      }
```

Updating Score

The last feature we are going to be adding today is the ability to update the score of a list item. Now, we have a decision to make- the click happens in the `ListItem` component which is a simple functional component. We could wire the component up with the connect method to give it access to the Redux store, but since it's direct parent is already connected, we can also pass the action creator down as a prop. Let's do the latter today.

In `List.js`, add a prop of `updateVote={this.updateVote}` to the rendered `<ListItem>` component in the `sortItems()` method.

Now define the `updateVote` method, which will need the itemId, currentScore, and value to add or subtract to the score.

```
  updateVote(itemId, currentScore, valueToAdd) {
     this.props.updateScore(itemId, currentScore, valueToAdd)
   }
```

Once again we are referencing an action creator we have yet to define - so let's go back to `actions/index.js` and write another function.

```
//actions/index.js

// ...

export const UPDATE_SCORE = 'update_score';

// ...

export function updateScore(itemId, currentScore, value){
  let score = currentScore + value;

  const request = axios.post(`${rootUrl}/item/${itemId}`, { score });

  return {
    type: UPDATE_SCORE,
    payload: request
  }
}
```

Now, back in `List.js` we need to import this action creator.

```
//List.js

// ...
import { addItem, updateScore} from '../actions';
// ...

function mapDispatchToProps(dispatch){
  return bindActionCreators({ addItem, updateScore }, dispatch)
}
```

And finally, now that we've got it all hooked up and are passing it down to the `ListItem` component

as a prop, let's call the action creator on click. Here's where the component should end up:

```
// ListItem.js

import React from "react";

const ListItem = ({ itemInfo, updateVote }) => {
  return (
    <li className="list__item">
      {itemInfo.item}
      <div className="item__scoreCard">
        <span className="scoreCard__score">{itemInfo.score}</span>
        <button onClick={() => updateVote(itemInfo._id, itemInfo.score,
          <img src={upvote} alt="thumbs up" />
          Upvote
        </button>
        <button onClick={() => updateVote(itemInfo._id, itemInfo.score,
          <img src={downvote} alt="thumbs down" />Downvote
        </button>
      </div>
    </li>
  );
};

export default ListItem;
```

Finally, after all that - let's add a case in our reducer to handle when the user fires the `updateScore` action creator.

```
import { FETCH_LISTS, ADD_ITEM, UPDATE_SCORE } from '../actions';

// ...
  case UPDATE_SCORE: {
        const newState = JSON.parse(JSON.stringify(state));

        // create some variables from our ajax response
        const listId = action.payload.data.item.belongs_to;
        const newItem = action.payload.data.item;
```

```
      const itemId = action.payload.data.item._id;

      // grab the array of list items
      let listItems = newState[listId].items;

      // search for the index of the list item we are updating
      const index = listItems.findIndex(item => item._id === itemId);

      // mutate the NEW COPY of state itself
      listItems[index] = newItem;

      return newState;
    }
```

Once again here we are using the `JSON.parse` method which will *not* work if we had non-serializable properties in our object.

Adding a new list

Just like yesterday- we're going to leave the functionality of adding a brand new list up to you! Can you write an action creator that will make a request to our API to add a list, and have the reducer create a new version of state without mutating it directly?

Some hints/reminders:

- Documentation for the API is here: Listed Docs

A repository of the completed code can be found Here.

back to classroom

HackerYou

hackeryou.com
info@hackeryou.com
github.com/hackeryou
@hackeryou