Writing our first Redux parts

Let's take the first steps in using Redux for our weather app

WE'LL COVER THE FOLLOWING

Our First Action

Let's write our first action! We'll start with the location field, since it's a very typical example. An action function in Redux returns an object with a `type` and can optionally also pass some data along the way. Our `changeLocation` action looks like this:

```
function changeLocation(location) {
  return {
    type: 'CHANGE_LOCATION',
    location: location
  };
}
```

This action thus has a type of `'CHANGE_LOCATION'` and passes along some data with the `location` property.

That's nice and all, but this won't change the state automatically. We have to tell Redux what to do when this action comes in, which we do in a so-called reducer.

A reducer is a simple function that takes two arguments, the current state and the action that was dispatched:

```
function mainReducer(state, action) {
  return state;
}
```

Right now, no matter what action comes in and what data it has the state will always stay the same – that's not quite optimal, as nobody will be able to work with the app! Let's change the `location` field in the state based on the data in the action with the `'CHANGE_LOCATION'` type.

```
function mainReducer(state, action) {
  switch (action.type) {
    case 'CHANGE_LOCATION':
      state.location = action.location;
      return state;
  }
}
```

What we're doing here is *mutating* the state. We assign `state.location` the value of `action.location`. This is discouraged by Redux because it introduces potential bugs and side effects. What we instead should be doing is *returning a new object* which is a copy of the state!

JavaScript has a handy function called `Object.assign`, which allows you to do that. Let's take a look at the solution first:

```
function mainReducer(state, action) {
  switch (action.type) {
    case 'CHANGE_LOCATION':
      return Object.assign({}, state, {
        location: action.location
      });
  }
}
```

By passing in a new, empty object (`{}`) as the first argument and the current `state` as the second one, we create a carbon copy of the state. The third argument of the function (`{ location: action.location }`) is *just the changes* argument of the function (`{ location: action.location }`) *is just the changes to our state*!

This creates a new object, meaning the state stays the same which is A+ behaviour and will keep us from a lot of bugs!

With a bit of glue this’ll already work! We should do two more small things to make this better: we should return the state unchanged if no action we want to handle comes in and we should use the initial state if state is undefined:

```
var initialState = {
  location: '',
  data: {},
  dates: [],
  temps: [],
  selected: {
    date: '',
    temp: null
  }
};

function mainReducer(state = initialState, action) {
  switch (action.type) {
    case 'CHANGE_LOCATION':
      return Object.assign({}, state, {
        location: action.location
      });
    default:
      return state;
  }
}
```

We’ll now need to `dispatch` this action when the location changes:

```
class App extends React.Component {
 fetchData = (evt) => { /* … */ };
 onPlotClick = (data) =>  { /* … */ };
 changeLocation = (evt) => {
   this.props.dispatch(changeLocation(evt.target.value));
 };
 render() { /* … */ }
});
```

> Don’t worry about where `this.props.dispatch` comes from for now, we’ll

> get to that!

Imagine `evt.target.value` is `"Sydney, Australia"`, this is what our global state is going to look like when we `dispatch` the `changeLocation` action:

```
{
  location: 'Sydney, Australia',
  /* …the rest stays the same… */
}
```

Tying it all together

Now that we understand the basic parts that are involved, let’s tie it all together!

First, we need to create a store for our state and provide the state to our root `App` component. The `store` combines all of the apps reducers and (as the name suggests) stores the state. Once the store is set up though, you can forget about it again since we’ll be using the state, but not the store directly!

> Redux allows us to create a single Store for the whole App

We do this in our main `index.js` file, and we’ll use the `createStore` function from the `redux` package and the `Provider` component from the `react-redux` package.

First, `import` those functions:

```
// index.js

/* … */
import ReactDOM from 'react-dom';

import { createStore } from 'redux';

import { Provider } from 'react-redux';

import App from './App.js';
/* … */
```

Then we need to create our store:

Then we need to create our store:

```
// index.js

/* … */
import App from './App.js';

var store = createStore();

ReactDOM.render(
/* … */
);
```

Lastly, we need to wrap our `App` component in the `Provider` and pass in the store:

Lastly, we need to wrap our `App` component in the `Provider` and pass in the store:

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

And that's it, our Redux integration is done! 🎉. Let's look at this in action

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

import App from './App';

let store = createStore();


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

Creating Actions

If you tried playing with the above example, you would have noticed that it doesn't do anything yet. Let's create an `actions.js` file and put our `changeLocation` action from above inside:

```
// actions.js

function changeLocation(location) {
  return {
    type: 'CHANGE_LOCATION',
    location: location
  };
}
```

We'll want to import it in other files, so we need to `export` it for that to work:

```
// actions.js

export function changeLocation(location) {
    return {
        type: 'CHANGE_LOCATION',
        location: location
    };
}
```

Awesome, we've got our first action – now we need to add our reducer!

Creating Reducers

Same deal as with the action, add a `reducers.js` file and export our previously written reducer from there:

```
// reducers.js

var initialState = {
  location: '',
  data: {},
  dates: [],
  temps: [],
  selected: {
    date: '',
    temp: null
  }
};

export default function mainReducer(state = initialState, action) {
  switch (action.type) {
    case 'CHANGE_LOCATION':
      return Object.assign({}, state, {
        location: action.location
      });
    default:
      return state;
  }
}
```

> We export the reducer by default since it'll be the only thing we're exporting from that file

That's our reducer done.

Enabling Store to use Reducers

Now we need to tell our store to use that reducer, so we `import` and pass it into the `createStore` call in the `index.js`:

```
// index.js

/* … */
import App from './App.js';

import mainReducer from './reducers';

var store = createStore(mainReducer);

ReactDOM.render(
/* … */
);
```

Connecting App to global redux state

Awesome, now everything's wired up except our `App` component! We need to connect it to the global redux state, which the `react-redux` module thankfully has a handy function for. Instead of exporting the raw `App` component, we export the `connect`ed component:

```
// App.js

/* …more imports… */

import { connect } from 'react-redux';

/* … */

export default connect()(App);
```

mapStateToProps

While this is nice, we also need to tell `connect` that it should inject the `location` field we have in our reducer into this component. We do this by passing in a function as the first argument that takes the entire state, and then we return what we want to inject as props into our component. (this automatically injects `dispatch` to run our actions, which is why we can use `this.props.dispatch` in the `App` component)

```
// App.js

/* … */

export default connect(function (state) {
    return {
        location: state.location
    };
})(App);
```

This function is called `mapStateToProps`, let's make that an external function so it's a bit clearer:

```
// App.js

/* … */

function mapStateToProps(state) {
    return {
        location: state.location
    };
}
```

```
export default connect(mapStateToProps)(App);
```

And that’s everything need to get our App to get the location from the Redux state! Let’s adapt our `App` to get the location from the props and see it in action

```
var initialState = {
  location: '',
  data: {},
  dates: [],
  temps: [],
  selected: {
    date: '',
    temp: null
  }
};

export default function mainReducer(state = initialState, action) {
  switch (action.type) {
    case 'CHANGE_LOCATION':
      return Object.assign({}, state, {
        location: action.location
      });
    default:
      return state;
  }
}
```

That’s everything needed to get the initial wiring done! Open this in your browser and change the location input, you should see the value adjusting – this means redux is working as expected!

Wiring up the rest

Let’s wire up some other actions, the goal here is to get rid of the entire component state of the `App` component! Let’s take a look at the selected date and temperature. The first we’ll write is two actions, `setSelectedDate` and `setSelectedTemp`, that pass on the value that they get passed in.

```
// actions.js

export function setSelectedDate(date) {
  return {
    type: 'SET_SELECTED_DATE',
    date: date
  };
}
```

```
export function setSelectedTemp(temp) {
  return {
    type: 'SET_SELECTED_TEMP',
    temp: temp
  };
}
```

Nothing fancy here, standard actions like the `changeLocation` one.

Let’s add those two constants to our reducer, and also adjust the initial state a bit to include those fields:

```
// reducers.js
export default function mainReducer(state = initialState, action) {
  switch (action.type) {
    case 'CHANGE_LOCATION':
      return Object.assign({}, state, {
        location: action.location
      });
    case 'SET_SELECTED_TEMP':
      return state;
    case 'SET_SELECTED_DATE':
      return state;
    default:
      return state;
  }
}
```

Now our reducer just needs to return the changed state for those actions:

```
// reducers.js

export default function mainReducer(state = initialState, action) {
  switch (action.type) {
    case 'CHANGE_LOCATION':
      return Object.assign({}, state, {
        location: action.location
      });
    case 'SET_SELECTED_TEMP':
      return Object.assign({}, state, {
        selected: {
          temp: action.temp,
          date: state.selected.date
        }
      });
      });
    case 'SET_SELECTED_DATE':
      return Object.assign({}, state, {
        selected: {
          date: action.date,
          temp: state.selected.temp
        }
      });
    default:
      return state;
  }
}
```

Now let's wire it all up again in our `App` component. The below example has three more actions (and constants and reducer cases) that need to be implemented here: `setData`, `setDates` and `setTemps`. I'll leave it up to you here to implement them, taking inspiration from our already implemented actions!

When you are done (or feel stuck), move to the next lesson and see all the actions & reducers implemented.

```
import React from 'react';
import { connect } from 'react-redux';

import './App.css';
import xhr from 'xhr';

import Plot from './Plot.js';

import {
  changeLocation,
  setSelectedTemp,
  setSelectedDate
} from './actions';


const API_KEY = "82f40c24bce69950c7aa3d09e07b391b";

class App extends React.Component {
  state = {
      data: {},
      dates: [],
      temps: []
  };

  onPlotClick = (data) => {
    if (data.points) {
      let number = data.points[0].pointNumber;
      this.props.dispatch(setSelectedDate(data.points[0].x));
      this.props.dispatch(setSelectedTemp(data.points[0].y));
    }
  };

  fetchData = (evt) => {

    evt.preventDefault();

    if (!API_KEY) {
      console.log('Enter your API_KEY and the enter location');
      return;
    }

    let location = encodeURIComponent(this.props.location);
    let urlPrefix = '/cors/http://api.openweathermap.org/data/2.5/forecast?q=';
    let urlSuffix = '&APPID=' + API_KEY + '&units=metric';
    let url = urlPrefix + location + urlSuffix;

    xhr({
      url: url
    }, (err, data) => {
      if (err) {
        console.log('Error:', err);
        return;
      }
      var body = JSON.parse(data.body);
      var list = body.list;
      var dates = [];
      var temps = [];
      for (var i = 0; i < list.length; i++) {
        dates.push(list[i].dt_txt);
        temps.push(list[i].main.temp);
      }

      this.setState({
        data: data,
        dates: dates,
        temps: temps,
      });

      this.props.dispatch(setSelectedTemp(null));
      this.props.dispatch(setSelectedDate(''));
    });
  };

  changeLocation = (evt) => {
    this.props.dispatch(changeLocation(evt.target.value));
  };

  render() {
    var currentTemp = 'not loaded yet';
    if (this.state.data.list) {
      currentTemp = this.state.data.list[0].main.temp;
    }
    return (
      <div>
        <h1>Weather</h1>
        <form onSubmit={this.fetchData}>
          <label>I want to know the weather for
            <input
              placeholder={"City, Country"}
              type="text"
              value={this.props.location}
              onChange={this.changeLocation}
            />
          </label>
        </form>
        {(this.state.data.list) ? (
          <div className="wrapper">
            {/* Render the current temperature if no specific date is selected */}
            <p className="temp-wrapper">
              <span className="temp">
                { this.state.selected.temp ? this.state.selected.temp : currentTemp }
              </span>
              <span className="temp-symbol">°C</span>
              <span className="temp-date">
                <p>The temperature on { this.props.selected.date } will be { this.props.selec
              </span>
            </p>
            <h2>Forecast</h2>
            <Plot
              xData={this.state.dates}
              yData={this.state.temps}
              onPlotClick={this.onPlotClick}
              type="scatter"
            />
          </div>
        ) : null}

      </div>
    );
  }
}

function mapStateToProps(state) {
        return {
                location: state.location,
    selected: state.selected
        };
}

export default connect(mapStateToProps)(App);
```