Exercise 11 – Flux and Redux

Objectives

1. To be able to convert an existing React Application into the Flux architecture.
2. To create a simple TodoList app using React and Redux.

Overview

The purpose of the first part of this exercise is to take the application from the previous exercise and restructure it to use the Flux architecture. This will involve modifying the components slightly (i.e. the views) and adding in the Dispatcher, a Store and Actions.

The second part of the exercise is an overview of the Redux framework that is used to implement the Flux architecture. This will walk through using reducers and actions to manage a Todo list.

PART 1 - Flux

Part 1.1 – Project Setup

1.1.1. Navigate to **EG11_FluxAndRedux/starter1** folder. This contains the stateless components for the application. This application files have been structured in a more usual hierarchy when developing. Take a look at the differences in the webpack.config.js file and see if you can discern how it is different from other projects you have seen. If you need it explaining, ask your instructor.

1.1.2. Check the output of the application before continuing so you are aware of the changes that are made because of the code you add.

Part 1.2 – Create a Store

In this part, we will set up and use a Store so that we can mimic the static application.

1.2.1. In the **js/development/stores folder**, create a new file called **ProductStore.js**
Note that this should be a JavaScript file and NOT a JSX.

1.2.2. Add the import for `{ EventEmitter }` from `events`.

1.2.3. Declare the class `ProductStore` and make it extend the `EventEmitter` class.

1.2.4. Add a `constructor` that calls `super()` and then sets `this.products` to be the array of items used in the **main.js** file of EG10.

1.2.5. Add another method to the class which returns the products array (suggest getAllProducts() as a name).

1.2.6. Close the class declaration and declare:

```
const productStore = new ProductStore;
export default productStore;
```

1.2.7. Save all files and check that there is no error output on the console. Resolve these now if there are, calling your instructor if you need help.

Part 1.3 – Set up the FilterableProductTable to hold state

Now we have a store with some data and a method to retrieve the data, the component to hold state for the application can now get this data, ready to pass it to its children.

1.3.1. In `FilterableProductTable`, add an import for the `ProductStore`.
1.3.2. Add a `constructor` to the class that calls `super()` and sets `products` to the return of the `ProductStore.getAllProducts()` method.
1.3.3. Add a `products` attribute to the `ProductTable` component that passes in the current state of products.
1.3.4. Check the console in your browser for errors, resolving any before you continue.

Part 1.4 – Populate the Product Table

As the products have now been passed to this child component, the logic and props to create the table, using the supplied `ProductCategoryRow` and `ProductRow` components, can now be added.

1.4.1. In the render method of `ProductTable` declare block variables of `rows` set to an empty array and `lastCategory` set to `null`.

1.4.2. Declare a `const` called `Products` and `map` this to the `products` passed in through `props` using a function as an argument that:

   a. Takes each `product` as an argument.

   b. Checks to see if the product's `category` is NOT the same as `lastCategory`.

      i. If true, add 2 elements to the `rows` array, the first being a `ProductCategoryRow` that has attributes of `category`, supplied by the product's `category` and a `key` (you can generate a random number for this) and a `ProductRow` that has attributes of the `key` being the product's `id` and the other `props` passed in as an object.

      ii. If false, create a `ProductRow` that has attributes of the `key` being the product's `id` and the other props passed in as an object.

   c. Sets `lastCategory` to by the current product's `category`.

Your code for this should look like:

```
const Products = this.props.products.map((product) => {
    if (product.category !== lastCategory) {
       rows.push(<ProductCategoryRow
           category={product.category}
           key={Math.floor(Math.random() * 2000)}/>);
       rows.push(<ProductRow key={product.id} {...product}/>);
    } else {
       rows.push(<ProductRow key={product.id} {...product}/>);
    }
    lastCategory = product.category;
});
```

1.4.3. In the `<tbody>` tags add in the expression for an evaluated `rows` array. (Use {}).

1.4.4. Open the `ProjectCategoryRow` component and ensure that the table heading uses the `category` that will be supplied by `props`.

1.4.5. Open the `ProductRow` component and in the `render` method add:

   a. A `const` that maps the object variables `price`, `name` and `stocked` to the object properties supplied by `props`.

   b. A `const` called `displayName` that adds a `style` in a `span` around `name` if `stocked` is `false` to make the text red (a ternary would be good here!).

   c. In the method's `return`, replace the static text `Name` and `Price` with the appropriate data.

Your code should look something like:

```
render() {
    const { price, name, stocked } = this.props;
    const displayName =
   (stocked) ? name : <span style={{color: 'red'}}>{name}</span>;
    return (
      <tr>
        <td>{displayName}</td>
        <td>{price}</td>
      </tr>
    );
}
```

1.4.6. Check the browser, you should now see a table populated with products, with a row that shows the category. Resolve any errors, if necessary before continuing

Part 1.5 – Making the Search Bar work – The Dispatcher

Now that the Store contains some data and is able to return this data, to make the Search Bar work, we need to add a Dispatcher, some Actions and further methods in the Store. The controller of all this is the Dispatcher and in this part we will add one to the application.

1.5.1. In the **components** folder, create a new file called **dispatcher.js**.

1.5.2. Import the `Dispatcher` class from `flux`.

1.5.3. Export a new `Dispatcher`.

1.5.4. Save the file, this is all that is needed for the dispatcher in this application.

Part 1.6 – Making the Search Bar work – Defining Actions

This Application will need 2 Actions defined. One will be the act of searching for products using the search box, the other will be displaying only those products that are in stock. This can be achieved by creating 2 functions in an Actions file.

1.6.1. In the **actions** folder, create a new file called **ProductActions.js**.

1.6.2. Import the `dispatcher` from the `dispatcher.js` file.

1.6.3. Create and export a function called `filterBySearch` that:

a. Takes an argument of `searchParameters`.

b. Calls the `dispatch` method on the `dispatcher`, passing in an object that contains a field called `type`, set to `FILTER_SEARCH` and the `searchParameters`.

1.6.4. Create and export a function called `filterByStock` that:

a. Takes an argument of `stockStatus`.

b. Calls the `dispatch` method on the `dispatcher`, passing in an object that contains a field called `type`, set to `FILTER_STOCK` and the `stockStatus`.

Part 1.7 – Making the Search Bar work – Registering and setting up the Store

Although the store can respond with all the data that is in the products array, it has no way of knowing if an Action has been called and even if it did, it has no way of responding to it. To enable all of this, the Store will be registered with the dispatcher to listen for broadcasts and have a callback function that will contain a switch statement to respond to different Action types.

1.7.1. Open the ProductStores.js file for editing.

1.7.2. In between the last 2 lines of code, insert the following, to register the Store with the Dispatcher:

```
dispatcher.register(productStore.handleActions.bind(productStore));
```

1.7.3. Inside the class definition, create the method `handleActions`. It should:

a. Take a parameter of `action`.

b. Have a `switch` statement, that uses the `action.type` parameter and has a case for each of the Action types defined in the **ProductActions.js** file

i. For the `FILTER_SEARCH` action, the case block should call a method called `filterBySearch`, that will be defined shortly, taking a parameter of `action.searchParameters`.

ii. For the `FILTER_STOCK` action, the case block should call a method called `filterByStock`, again to be defined, taking a parameter of `action.stockStatus`.

iii. The `default` case should simply `break;` If the Store does not need to respond to the broadcast Action type then it will do nothing.

1.7.4. Add an empty array called `filteredProducts` to the constructor – this will be used to return the filtered results.

1.7.5. Add a method to return the `filteredProducts` array.

1.7.6. Define the `filterBySearch` method in the class. It should:

a. Receive the `searchParameters`.

b. Empty the `filteredProducts` array.

c. Use a `forEach` loop on the `products` array that executes a function that takes a parameter of the `product` and then checks to see if the `searchParameters` is part of the product `name` and pushes the product to the `filteredProducts` array if it is.

d. Finally, the method should `emit` the `change`.

Your code for this method should look like:

```
filterBySearch(searchParameters) {
    this.filteredProducts = [];
    this.products.forEach((product) => {
      if(product.name.indexOf(searchParameters) !== -1) {
        this.filteredProducts.push(product);
      }
    });
    this.emit('change');
  }
```

1.7.7. Define the `filterByStock` method in the class. It should:

a. Receive the `stockStatus`.

b. Empty the `filteredProducts` array.

c. Use a `forEach` loop on the `products` array that executes a function that takes a parameter of the `product` and then checks to see if the `stocked` variable matches the `stockStatus` and that the `stockStatus` is `true`, pushing the `product` to the `filteredProducts` array if these conditions are met.

d. Finally, it should `emit` the `change`.

Your code for this method should look like:

```
filterByStock(stockStatus) {
    this.filteredProducts = [];
    this.products.forEach((product) => {
      if(product.stocked === stockStatus && stockStatus === 
true) {
        this.filteredProducts.push(product);
      }
    });
    this.emit('change');
  }
```

Part 1.8 – Making the Search Bar work – Making the FilterableProductTable respond to Store changes

Lifecycle methods will be used to register an event listener and then call a method to update the state of the `FilterableProductTable` component. The event listeners will subscribe to the Store's change message and then call a method to handle this event.

1.8.1. In **FilterableProductTable.jsx** add the `componentWillMount` method. It should call a 'private' method `_onChange` from the `ProductStore` `on` `change` event.

1.8.2. Add the `componentWillUnmount` method. It should turn the event listening off.

(Use `removeEventListener` here!).

1.8.3. Add the `_onChange` method. It should use `setState()` to set the `products` array to the array returned by the `ProductStore`'s `getFilteredProducts` method.

1.8.4. In the `constructor`, bind the `_onChange` method to `this`.

Part 1.9 – Making the Search Bar work – Passing props to the SearchBar

The `SearchBar` needs to react to changes in the inputs by the user. To do this we will add some more state, attributes and methods in the `FilterableProductTable` component and then refs and then have 2 handling functions that will be attributes that will be added in the `FilterableProductTable` component.

1.9.1. In **FilterableProductTable.jsx** add `filterText` as an empty string and `inStockOnly` set to `false` to the initial state in the `constructor`.

1.9.2. In **FilterableProductTable.jsx** modify the `SearchBar` component so that:

a. It has an additional attribute of `filterText`, set to the current `state`.

b. It has an additional attribute of `inStockOnly`, set to the current `state`.

c. It has an additional attribute of `onUserSearchInput` that will call a (yet undefined) method of `handleUserSearchInput`.

d. It has an additional attribute of `onUserStockInput` that will call a (yet undefined) method of `handleUserStockInput`.

e. In the `constructor`, `bind` both (undefined) method calls to `this`.

1.9.3. In **SearchBar.jsx**, add a constructor to receive props.

1.9.4. In **SearchBar.jsx**, modify the `text` input so that:

a. It has an additional attribute of `value` set to `filterText` passed in by `props`.

b. It has an additional attribute of `ref` which is an anonymous function receiving `input` and setting `this.filterTextInput` to `input`

c. It has an additional attribute of onChange which calls a (yet undefined) method of handleSearchChange.

1.9.5. In **SearchBar.jsx**, modify the `checkbox` input so that:

a. It has an additional attribute of `checked` set to `inStockOnly` passed in by `props`.

b. It has an additional attribute of `ref` which is an anonymous function receiving `input` and setting `this.inStockOnlyInput` to `input`

c. It has an additional attribute of `onChange` which calls a (yet undefined) method of `handleStockChange`.

1.9.6. In the constructor, `bind` `handleSearchChange` and `handleStockChange` to `this`.

1.9.7. Create a method called `handleSearchChange` that calls `this.props.onUserSearchInput` with an argument of `this.filterTextInput.value`.

1.9.8. Create a method called `handleStockChange` that calls `this.props.onUserStockInput` with an argument of `this.inStockOnlyInput.checked`.

Part 1.10 – Making the Search Bar work – Make FilterableProductTable call Actions on Search

The final part of this exercise is to define the methods that will cause the filtering to be done depending on the user's input. To do this, the 2 undefined methods from the last part will be created. These will call the Action methods which dispatch the instructions to the Store which then updates, triggering the change event to be emitted and subsequently the state of `FilterableProductTable`, causing `ProductTable` and its associated components to update meaning ultimately the view the user sees updates! Phew!

1.10.1. In **FitlerableProductTable.jsx**, add an `import` for *all* methods in `ProductActions` `as ProductActions`.

```
import * as ProductActions from '../actions/ProductActions';
```

1.10.2. In **FilterableProductTable.jsx**, define the method `handleUserSearchInput`. It should:

a. Receive an argument of `filterText`.

b. Call `setState` to set `filterText`.

c. Call `ProductActions.filterBySearch`, passing in `filterText`.

1.10.3. In **FilterableProductTable.jsx**, define the method `handleUserStockInput`. It should:

a. Receive an argument of `inStockOnly`.

b. Call `setState` to set `inStockOnly`.

c. Call `ProductActions.filterByStock`, passing in `inStockOnly` ***IF*** `inStockOnly` is `true`, otherwise call `setState` to make `products` be the result of `ProductStore.getAllProducts()`.

Saving all files and viewing in the browser should now present you with a working application, implementing the Flux architecture!

Part 2 – Redux

Part 2.1 – Project Setup

Navigate to **EG11_FluxAndRedux/starter2** folder. This contains the minimum code required to display a ToDo List and to have an input box and an add button. Currently, the application does little more than display a dummy ToDo. The aim of the exercise is to use Redux to create an application which allows the addition of more ToDos.

2.1.1. Examine the files **main.js**, **App.jsx** in the root folder and the *3* **jsx** files in the **components** folder. Be sure that you understand how the current application works before continuing.

2.1.2. Examine the **package.json** file and note that the `redux` and `react-redux` packages have already been installed as part of the set-up of the project. These will be used as the application is created.

Part 2.2 – ACTIONS

In this part, the actions for the application will be defined. For simplicity, there will be a single action, `ADD_TODO`.

2.2.1. In the **actions** folder, create a new file called **actions.js**.

2.2.2. Export a `const` called `ADD_TODO` set to the string `‘ADD_TODO’`.

2.2.3. Declare a new variable called `nextTodoId` and set it to `0` (zero).

2.2.4. Export a `function` called `addTodo` that:

a. Takes an argument of `text`.

b. Returns an object with `type` set to `ADD_TODO`, an `id` set to a ***post-incremented*** `nextTodoId` and `text` set to `text`.

2.2.5. Save the file.

Part 2.3 – REDUCERS

The reducers specify the changes that will be made in the application. For simplicity, there will be a single reducer but for future reference a combined reducer has been included to allow additional reducers to be included. It is worth noting that the combined reducer is usually in its own file, importing the other reducers from their own files.

2.3.1. In the **reducers** folder, create a new file called **reducers.js**.

2.3.2. Import the function `combineReducers` from the `redux` package.

2.3.3. Import the `const` `ADD_TODO` from the **actions.js** file.

2.3.4. Create a function called `todo`. This will be a reducer function and therefore requires arguments of `state` and `action`.

2.3.5. In `todo`, define a `switch` statement that:

   a. Takes an argument of `action.type`.

   b. Has a `case` for `ADD_TODO` that returns an object with `id` set to `action.id` and `text` set to `action.text`.

   c. Has a `default` case that returns `state`.

2.3.6. Create a function called `todos`. This will also be a reducer function and therefore requires arguments of `state` and `action`. In this case, by default, `state` should be an empty array.

2.3.7. In `todos`, define a `switch` statement that:

   a. Takes an argument of `action.type`.

   b. Has a `case` for `ADD_TODO` that returns an array with a *spread* of `state` and a call to the `todo` function, supplying `undefined` and `action` as arguments.

   c. Has a `default` case that returns `state`.

2.3.8. Define a `const` called `todoApp` and set this to the `combineReducers` function, supplying an *object argument* containing `todos`.

2.3.9. Add the line `export default todoApp`.

2.3.10. Save the file.

Part 2.4 – The STORE

The Store will hold the Todo applications state. Currently, an array is supplied by the **App.jsx** file but this will be replaced with the Store that will be created here.

2.1.1. In the **stores** folder, create a new file **TodoStore.js**.

2.1.2. Import `createStore` from `redux`.

2.1.3. Import `todoApp` from `reducers`.

2.1.4. Export a `const` `store` set to the return of the `createStore` function supplying `todoApp` as an argument.

2.1.5. Save the file.

Part 2.5 – Wrap the App in a Provider

To allow the whole application to access the Store, the App component will be wrapped in a special React-Redux component called Provider. The Store will be passed in to this component allowing all child components to access it through props.

2.5.1. Open **main.js**.

2.5.2. Add an import for the `Provider` component from the `react-redux` package.

2.5.3. Import the `store` from the `TodoStore` file.

2.5.4. In the `ReactDom.render` method, wrap the `<App />` component in a `<Provider>` component, suppling the `store` as an attribute (`store={store}`).

2.5.5. Save the file and check the output for errors on the console, resolving them before moving on.

Part 2.6 – Make the App component display the Todos and allow new ones to be added

The final part of this exercise is to make the magic happen! First the application needs to know about the Todos that are already listed. Secondly, it needs to know about the `ADD_TODO` action so that it can pass the contents of the text input to the action to update the Store.

2.6.1. Open the **App.jsx** file.

2.6.2. Add an `import` for the `connect` function from `react-redux`.

2.6.3. Add an `import` for the `AddTodo` action from the `actions` file.

2.6.4. ***Remove*** the `export default` from the `class` definition – this will be replaced by a call to the redux connect method later.

2.6.5. To make the current Todos visible:

a. Change the `const` in the `render` method to an ***object definition*** of `dispatch` and `visibleTodos` set to `this.props`.

   b. Change the attribute supplied to the `TodoList` component to `visibleTodos`.

2.6.6. Outside of the class, define a new function called `select` that:

   a. Receives an argument of `state`.

   b. Returns an object containing `visibleTodos` set to `state.todos`.

2.6.7. Add the line `export default connect(select)(App);` to the end of the file.

2.6.8. Save all files and check the output has no errors on the console. If it has, resolve before continuing.

2.6.9. To be able to add Todos to the list:

   a. Change the `onAddClick` attribute of the `AddTodo` component so that:

      i. The argument of the arrow function is `text`.

      ii. The statement in the curly braces is a call to the `dispatch` function supplying a call to `addTodo` with an argument of `text`.

2.6.10. Save all files and check that the application works.