Exercise 11 – Using React with Redux

Objectives

To use Redux state management to retrieve and deliver data into an application.

Overview

The Star Wars fan site created as part of the last exercise used static data, held within the components to provide the content for the pages. In reality, this data would be held on a server and delivered to the application when requested. A state management tool, such as Redux is useful for this. If our application needed to hold any state, then Redux could also be used here. In this exercise, you will:

- Use a development tool called **json-server** to provide a fully working mocked RESTful service to provide data about films and actors;
- Populate and provide **Redux** store with data from **json-server**;
- Create **reducers** to handle requests for actors and films;
- Define a number of **actions** for actors and films;
- Create **container** components for **ActorDetails** and **FilmDescription**;
- Make container components make calls for data and receive data from the store.

Exercise Instructions

Check that the starter project runs correctly

1. In the command line, navigate into the **starter** folder for **EG11**.
2. Run the command:

```
npm install
```

3. Now run the command:

```
npm start
```

Verify that the browser outputs a single page application, the same as that observed at the end of the last exercise.

Provide a mock RESTful server

1. In a *separate* command line (i.e. leaving the React project running), navigate to the **data** folder.

json-server has been installed globally on the computer already. It is an **npm** package and it provides a fully functioning mock **RESTful** server. It runs from the command line and only needs a properly configured **.json** file to deliver its data. By default, it runs on port **3000**, but since our **React** application is *already using this*, we will change this when we start it up.

2. Run the following command to start **json-server** with the **starWars.json** file in the **data** folder and directing it to **port 8080**:

```
json-server starWars.json -p 8080
```

This should spin up **json-server**.

3. Check that **json-server** is correctly serving data at the following addresses:
   - **http://localhost:8080/actors** - all of the actors in the starWars.json file
   - **http://localhost:8080/films** - all of the films in the starWars.json file
   - **http://localhost:8080/actors/HF** - Harrison Ford's details
   - **http://localhost:8080/films/SW4** - Star Wars IV - A New Hope's details

If your server is working correctly, you may move on. If not, ask your instructor for help.

Create Actions

Actions define what our requirements for data are. In this simple application, we will write actions to get an individual piece of data from the appropriate section of the json file. These actions will be called from the Container component we will create to connect to Redux. These are purely JavaScript files and need no imports.

Actions for the Actors

1. In the **EG11 starter/src** folder create a folder called **actions**.
2. Inside **actions**, create a file called **actorActions.js**.

Each action will have a type property. Best practice is to create a **const** for these types. As we are ultimately making an HTTP request, we will have an action to:

- Initiate the request;
- Deal with a successful request;
- Deal with a failed request.

Defining constants

3. Define **export**ed **const**ants and set them to their string equivalents:
   - **GET_ACTOR_BEGIN**
   - **GET_ACTOR_SUCCESS**
   - **GET_ACTOR_FAILURE**
4. Create a **const** to define the **url** for the **actors** on **json-server**

Defining action functions

5. Create an **export const** called **getActorBegin** as an *arrow function* that takes *no arguments* and **returns** an **object** with a *key* of **type** and a **value** of **GET_ACTOR_BEGIN**.
6. Create an **export const** called **getActorSuccess** as an *arrow function* that:
   - Takes an argument of **actor**;
   - **Returns** an **object** with:
     - A *key* of **type** and a *value* of **GET_ACTOR_SUCCESS**;
     - A *key* of **payload** and a *value* of an *object* that has **actor** as a *key-value pair*.
7. Create an **export const** called **getActorFailure** as an *arrow function* that:

- Takes an argument of **error**;
- **Returns** an **object**:
    - A *key* of **type** and a *value* of **GET_ACTOR_FAILURE**;
    - A *key* of **payload** and a *value* of an *object* that has **error** as a *key-value* pair.

These actions need to be called from somewhere that *dispatches* them. We will create a function called **getActor** and supply this with the **id** to retrieve.

Dispatching actions and obtaining results

8. Create an **export const** called **getActor** as an *arrow function* that:

    - Takes **actorId** as an argument;
    - **Returns dispatch** as an *arrow function* that:
        - Calls **dispatch** with an argument of **getActorBegin()**;
        - **Returns** the *result of a fetch* of the *defined URL* with a sub-location of the **actorId** and then:
            - Has an *arrow function* that takes **response** and *returning* **response** *passed to a function* (as yet unwritten) called **handleErrors**;
            - Has an *arrow function* that takes **response** *returning* the result of running the **json()** function on it;
            - Has an *arrow function* that takes the *returned JSON* that calls **dispatch** with an argument of calling **getActorSuccess** with has its own *argument of the JSON* and then **returns** *the JSON*;
            - If an **error** is *caught*, this should be passed into an *arrow function* that calls **dispatch** with an argument of calling **getActorFailure** with the **error** as its argument.

This function has called a function called **handleErrors**. We will use this to check to see if the **fetch** returns a *result we do not want*. Since a Promise will resolve even if an undesired status is returned (404, 503, etc), we need to check that **response** contains the data we want it to.

9. Declare a **const** called **handleErrors** as an *arrow function* that:

    - Takes **response** as an argument;
    - Check to see **if** the **ok** property of **response** is **false** *OR* the **status** property of **response** is *not equal to 200*, throwing an **Error** with **statusText** if this is the case;
    - Returns **response**.

10. Save the file.

This concludes the actions! The application needs to know how to handle the dispatches that actions can produce. This is the job of reducers.

Creating the Actor Reducer

A reducer takes the actions and reduces these to returning the next state required by the application due to the action called.

1. In **src**, create a new folder called **reducers**.
2. In **reducers**, create a new file called **actorReducer.js**.
3. Add **imports** for the **consts** in **actions/actorActions.js** for **GET_ACTOR_BEGIN**, **GET_ACTOR_FAILURE** and **GET_ACTOR_SUCCESS**.
4. Create a **const** called **initialState** and set this to be an *object* with *key-value pairs*:
   - **actorToDisplay** set to an *empty* **object**;
   - **loading** set to **false**;
   - **error** set to **null**;

A reducer function should take a state and an action and return the next state.

5. Define an **export const** called **actorReducer** as an *arrow function* that:
   - Takes **state** set to **initialState** *by default* and an **action**;
   - Has a body that **switches** on the **type** *of* **action** and for:
     - **GET_ACTOR_BEGIN**: **return** an *object* with a *spread* of **state**, **loading** as **true** and **error** as **null**;
     - **GET_ACTOR_FAILURE**: **return** an *object* with a *spread* of **state**, **loading** as **false**, **error** as **action.payload.error**, **actorToDisplay** an *empty* **object**;
     - **GET_ACTOR_SUCCESS**: **return** an *object* with a *spread* of **state**, **loading** as **false** and **actorToDisplay** as **action.payload.actor**;
     - **default**: **return state**.
6. Save the file.

We still don't have a store to retrieve data from. To do this, we need to create a store.

Creating the Store

To create the store, we need to use Redux. We need to add Redux to our project dependencies.

1. In a *new command line* (leave webpack and json-server running), navigate to the **EG11 starter** folder.
2. Add **redux** to the list of dependencies using:

```
npm i redux --save
```

3. In **src**, create a new file called **store.js**.
4. **Import** the required Redux functions here, namely **combineReducers**, **createStore** and **applyMiddleware** from **redux**.

To be able to work with asynchronous data in our store, we need to use some middleware. The chosen middleware for this exercise is **thunk**.

Redux Thunk middleware allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met. The inner function receives the store methods dispatch and getState as parameters.

https://github.com/reduxjs/redux-thunk

5. Navigate to the project on the command line and install **redux-thunk** using:

```
npm i redux-thunk --save
```

6. In **store.js**, import **thunk** from the package just installed.
7. Import the **actorReducer** from the **actorReducer** file.
8. Declare a **const reducers** and set this equal to a call to the **combineReducers** function, passing in an object that contains key-value pair **actorReducer**.

Note: we can add to this list of reducers as and when we create them.

9. Create a **const store** that is set to the result of a **createStore** call, passing in **reducers** and also a call to the **applyMiddleware** function that takes **thunk** as an argument.
10. **Export store** as a **default**.
11. Save the file.

Provide the application with the store

Now we have a Redux store, we need to let the React application know that it exists and provide it to it! This is generally done when we initialise our React application. We are going to surround our App component with a React-Redux component called **Provider**, that has a **store** property.

1. Install **react-redux** as a *project dependency* using **npm**.
2. Open **index.js** from the **src** folder.
3. **Import** the *non-default* Component **Provider** from **react-redux**.
4. **Import** the **store** from **store.js**.
5. In the **render** method, wrap the **App** component with a **Provider** component. The **Provider** component should have an attribute **store** set to the *imported value*.
6. Save the file.

Create a container component

So that the actual component logic is not polluted with any code needed to connect to Redux, a pattern that uses a container for this logic is often used. The purpose of the container is to map the values from Redux to the props of the container, thereby passing the display component data through its props.

1. In **src**, create a new folder called **containers** and a file called **VisibleActors.jsx**.
2. Import the following:
    - **connect** from **react-redux** (non-default);

  - **ActorDetails** component;
  - **getActor** method from **actorActions** (non-default);

Mapping data in the component

1. Create a **const** called **mapStateToProps** as an *arrow function* that:
   - Takes an argument of **state**;
   - Returns an *object* with key-value pairs:
     - **actor** provided as **state.actorReducer.actorToDisplay**;
     - **loading** provided as **state.actorReducer.loading**;
     - **error** provided as **state.actorReducer.error**.
2. Create another **const** called **mapDispatchToProps** as an arrow function that:
   - Takes an argument of **dispatch**.
   - Returns an *object* with key-value pair:
     - **update** provided as an *arrow function* that takes **id** as an argument *returning* a call to **dispatch** that has an argument of the **getActor** method supplied with **id**.

Connecting the components

1. Create a **const** called **VisibleActor** that is set to a **connect** function call with **mapStateToProps** and **mapDispatchToProps** as its *first set of arguments* and **ActorDetails** as its second set.
2. **Export VisibleActor** as a **default**.
3. Save the file.

Now we have the data in a container that is able to retrieve the data we need in our application and supply it to the display component through props.

Rendering the container component

As the container component provides the display component with data, it's the container that needs to be rendered in the Actors component.

1. Open **Actors.jsx** from the **src/actors** folder.
2. Change the *rendered component* in the **Route** where the **actorId** is used as part of the path to **VisibleActor**.
3. *Remove* the **import** for **ActorDetails** and *add* the **import** for **VisibleActor**.
4. Save the file.

Using data in the display component

What is needed next is to make the **ActorDetails** component use the data supplied to it by the **VisibleActor** container component. This will be done by a mixture of evaluating internal state and props, responding to conditions in lifecycle methods and rendering the appropriate content due to errors, loading or an actor being present. We also need to ensure that this component updates when a new actor is selected.

Prepare the component

1. Open **ActorDetails** from the **src/actors** folder.

For speed of the exercise, the **ActorDetails** component has been converted into a class.

2. Remove the **actors** array declared before the class.
3. Remove the lines of code that **filter** the **actors** array in the **render** method to produce an **actorToDisplay**.

We will replace this code with calls to Redux actions that deliver the required data.

The initial render

In the first instance, we want the component to mount with the actor that has been clicked initially. A good place to do this is in the **componentDidMount** method. Here we will call the **update** method that was defined in the **mapDispatchToProps** function in the container.

1. Add a **componentDidMount** method to the component.
2. In the method body call the **update** method from **props**, passing in the **actorId** present in **prop**'s **match.params** object.

Now the render method will be modified so that it will examine the props supplied for errors and loading conditions so that alternate returns can be provided.

3. In the **render** method, add a **const** that *deconstructs* **props** as **error**, **loading** and **actor**.
4. Set a **const actorToDisplay** to **actor**.
5. Check for an **error** and if present, **return** a **div** that has the text **"Error!"** and displays the **error**'s **message**.
6. Check for **loading** and if present, **return** a **div** with the text content **"Loading"**.
7. The actual return section of the render method is unchanged.
8. Save the file.

You can now return to your browser, navigate to the Actors section and select an actor. Their details should be shown. However, if you click on another actor, their details are not shown, even though the URL changes. This is because we haven't updated the component - there is no call to an action to dispatch a new actor and subsequently update the props that affect the DOM. As such, the component is not re-rendered as there is apparently no need to.

Making the component update

This is more complicated that it may first appear. Obviously, when we click on a link, the component's lifecycle methods are initiated and if we simply run the same update method as called in the mounting method, we will enter into an infinite loop of updating. We need to devise a way of knowing which actor is currently rendered and if that is different from the actor in the next set of props received and render if it is different. Time to use **getDerivedStateFromProps**.

9. Add a **constructor** and initialise **state** with 3 values, **prevActorId**, **prevError** and **prevLoading** all set to **null**.

- Add the **static** method **getDerivedStateFromProps** that:
    - Takes arguments of **nextProps** and **prevState**;
    - Checks the result of the *boolean* operation:
        - The **actorId** in the **next** set of **props** from the **match.params** object *is not equal to* **prevState**'s **actorId**;

OR

        - **error** in **nextProps** doesn't equal **error** in **prevState**;

OR

        - **loading** in **nextProps** doesn't equal **loading** in **prevState**.

10. If this *boolean* operation returns **true** return an *object* with key-value pairs:

- **prevActorId** set to **nextProps.match.params.actorId**;
- **prevError** set to **nextProps.error**;
- **prevLoading** set to **nextProps.loading**.

This now will change the **state** of the component depending on whether the **props** used change or not. Now when the component updates, we can initiate a call to **update** from **props** to get data for a change in **actorId**.

11. Add a **componentDidUpdate** method to the component. It should:

- Check the value of the *boolean* expression:
    - **state**'s **prevError** is **false**;

AND

    - **state**'s **prevLoading** is **false**;

AND

    - **state**'s **prevActorId** is *not equal* to the **id** of the **actor** object passed in by **props**.
- If this is **true**, then the **update** method from **props** should be called with an argument of **state**'s **prevActorId** value.

12. Save the file.

Check the browser and you should see that the application will now update the when you choose different actors from the menu.

What about the Films?

The process is exactly the same! If you have time, fly 'Solo' again!

"*Remember the Force* [and the solution code] *will be with you, always!*"