Exercise 9 – Single Page Applications

Objective

To be able to use React Router to create a single page application.

Overview

The purpose of this exercise is to practice using React Router along with many of the tools and techniques covered so far to create a single page application. The premise for the application is a fan site for the Star Wars franchise (the original trilogy only) but alternative data could be used if this does not appeal! The application should have:

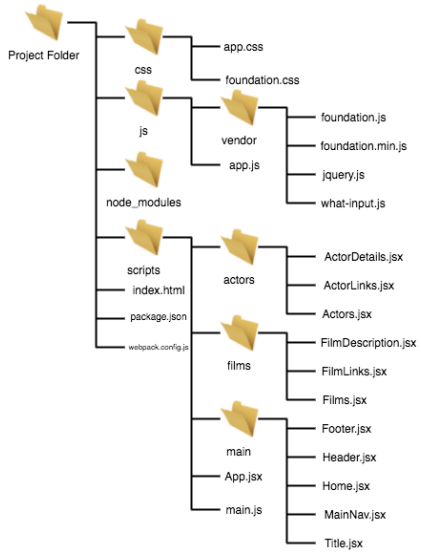
1. A navigation bar that is always present throughout the application
2. Home page content, constituting an introduction to the website
3. A page for films that has links to the original 3 films. Clicking on the link should:
    - Reveal the name of the film
    - Display a synopsis of the film
4. A page for actors that has links to the 3 biggest stars. Clicking on the link should:
    - Reveal the name of the actor and a reference to the character that they played
    - A short biography

Styling and other functional JavaScript (such as Foundation and jQuery) has been provided to allow you to focus on the React part of the exercise.

Part 1 – Project Setup

1.1. You may create your own project set up for this project or you can use the **EG09_SinglePageApplications/starter** folder.

1.2. The folder structure for the project is as follows:



Part 2 – Check React Router is in the project

2.1. Open the **package.json** file and check that **react-router** is installed.

2.2. If it isn't, install **React Router** through **npm** using the command:

```
npm install –-save react-router
```

Part 3 – Check the project set-up works

It is a good idea to check that a basic set up works before moving to a more complicated application.

3.1. In **scripts/main.js**, add the code to render an **App** component.

3.2. In **scripts/App.jsx**, add the code to render a simple component that displays "Hello World", or similar.

3.3. Run the project and check the output.

Part 4 – Set up the routing for the application

The application will have three routes set up, one for Home, one for Films and one for Actors. Home will also be set up as the IndexRoute. These will be defined within the **scripts/main.js** file.

4.1. In the **scripts** folder, open **main.js**.

4.2. Add an `import` for the following items from `react-router`:

```
{ Router, Route, IndexRoute, browserHistory }
```

4.3. Within the ReactDOM's render method, Set the Router to have an history attribute of browserHistory

4.4. The application will have routes for:

- The `path` of `/` which will be set to display the `App` component

- An `IndexRoute` set to display the `Home` component
- A `Route` to `/films` set to display the `Films` component
- A sub `Route` within `/films` using the parameter `filmName` and displaying the component `FilmDescription`
- A `Route` to `/actors` set to display the `Actors` component
- A sub `Route` within `/actors` using the parameter `actorName` and displaying the component `ActorDetails`

4.5. Add the necessary imports for all components used in the application's routes.

Part 5 – The App Component

To make a truly React interface, large components should be split down into smaller components. The App component is a prime candidate for this as it is composed of a Header section, a Main section and a Footer section. Creating the Header and Footer as separate components increases their reusability.

5.1. In **App.jsx**, replace the current render method with a `<div>` that wraps:

- A `<Header />` component
- A `<main>` element with a `className` of `row` that has the contents of any `children` (`{this.props.children}`)
- A `<Footer />` component

5.2. As `<Header />` and `<Footer />` will be functional components, add their imports including `{ }` around the component name.

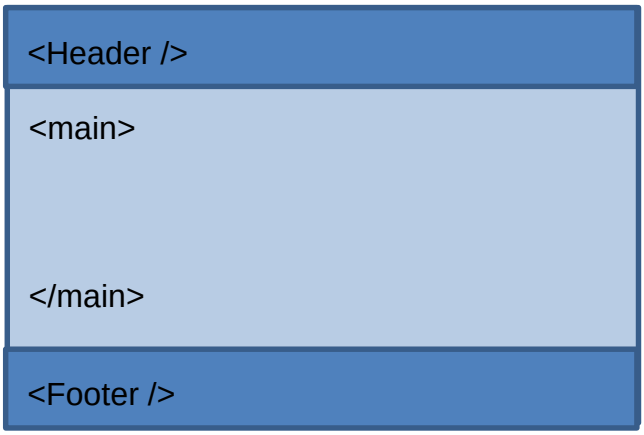


Figure 1 - The App Component Composition

5.3. Save the **App.jsx** file.

Part 6 – The Header, Title and MainNav Components

The <Header /> itself is a complex component that is to be laid out as shown:

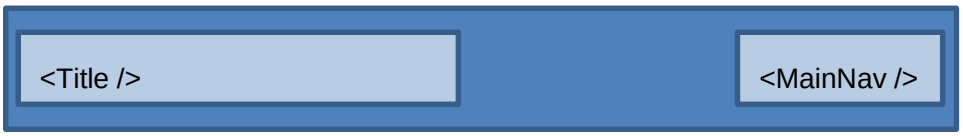


Figure 2 - Composition of the <Header /> Component

The <Title /> component will be functional and will contain the application's title and some code to make the application responsive. The <MainNav /> component will also be functional and will contain code to create the navigation for the main part of the application.

6.1. The <Title /> component will contain the application title and some code to make the application responsive. The <MainNav /> component will cCreate the component as a `function` that is exported. It should be wrapped in a `<div>` that has a `className` of `top-bar`.

6.2. Import the Title and MainNav components as functions from their JSX files.

6.3. Save the **Header.jsx** file.

The `<Title />` component is functional and should be set up as so. It should be saved in the scripts/main folder. The layout for this component is as follows:

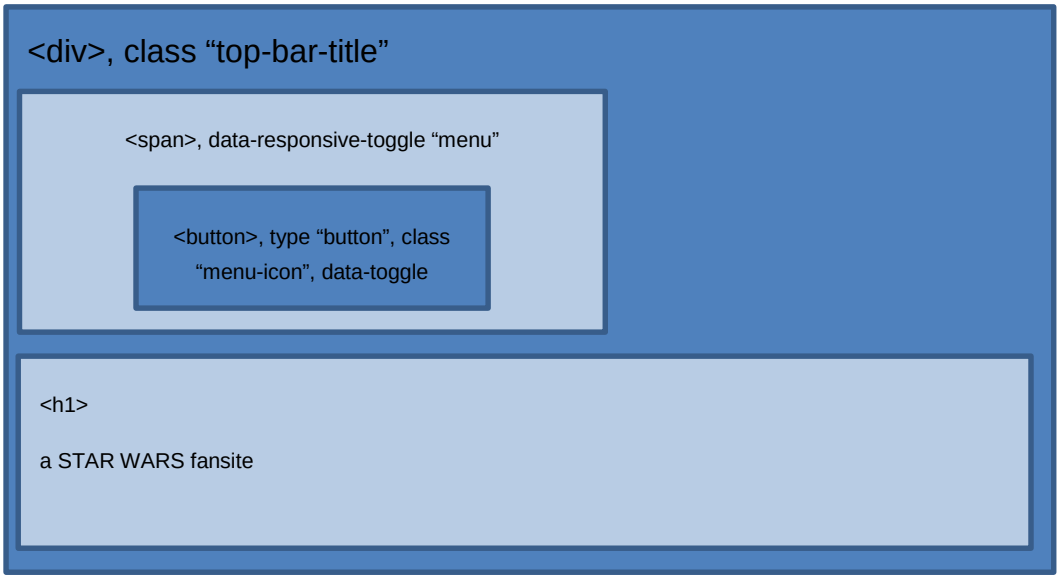


Figure 3 - The Title Component Composition

6.4. Create the function to return the html to create the layout above.

The final component of <Header /> is the MainNav component. It could be argued that this itself could be split into components, but for the sake of simplicity in the exercise it has been left 'as is'. The component layout is as follows:

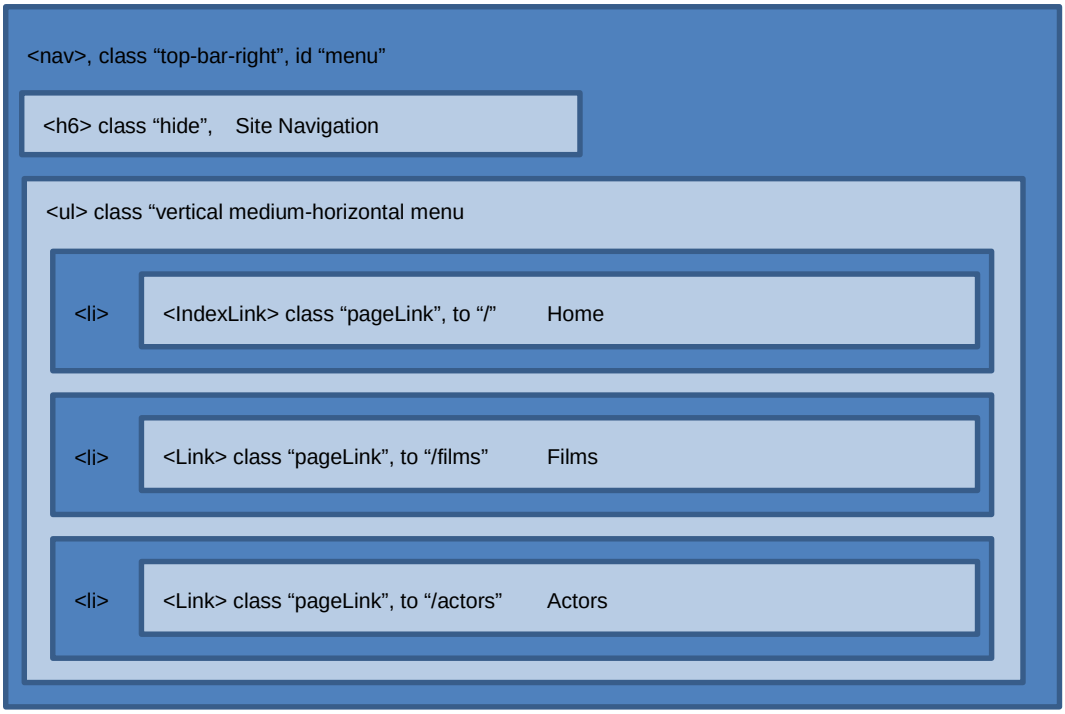


Figure 4 - The MainNav Component Configuration

6.1. Create the `<MainNav />` component as a function to return the layout as described above.

6.2. `import` the required React-Router components (`IndexLink` and `Link`).

6.3. Save the **MainNav.jsx** file.

Part 7 – The Footer Component

The `<Footer />` should again be a functional component that simply returns a `<footer>` element with the text ‘*Made for fans by a fan*’.

Removing so far unused routes/imports/components from the code will allow you to see the application thus far, if you so wish to.

Part 8 – The Home Component

The <Home /> component is another component that could be split into other components, but since it has no state, it has been left as is. This component could be classed as functional but since it is used in main.js and not as a subcomponent of another, it needs to have a render method and will therefore be declared as a class.

The layout of this component is as follows:

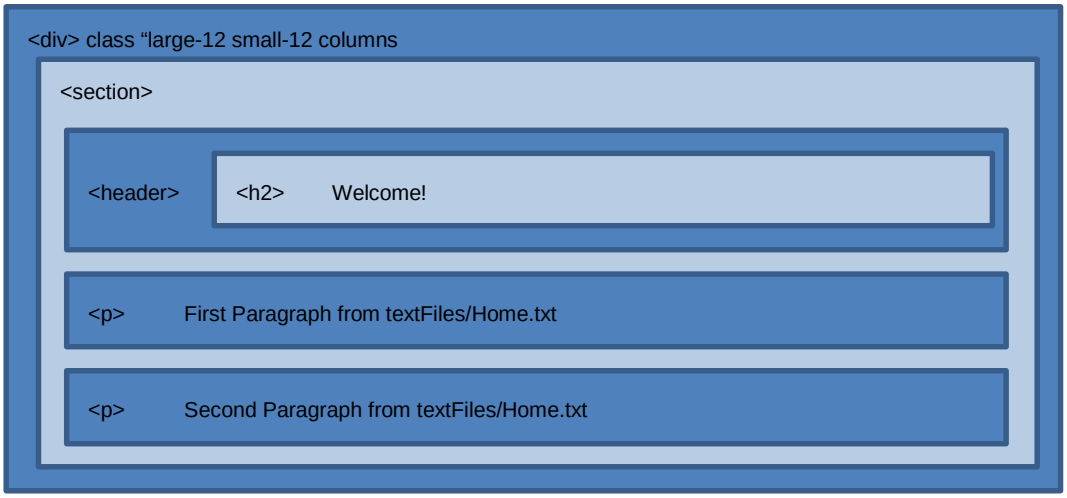


Figure 5 - Home Component Composition

8.1. Create the component as shown. Text for the paragraphs can be found in the **Home.txt** file which has been provided in **starter/textFiles**.

8.2. Save the **Home.jsx** file.

Part 9 – Films and Actors Components

The way in which these components are created is relatively similar. There is a JavaScript object provided for each, which will act as the data source for the application. These can be found in the **starter/textFiles** folder in **actors.txt** and **films.txt**. Essentially, the main component will use the subroutes defined in **main.js** to render child components that have the requested information. As with the `<Home />` component, they will be created as *classes* as they need to have an explicit `render` method defined.

Both main components are laid out identically:

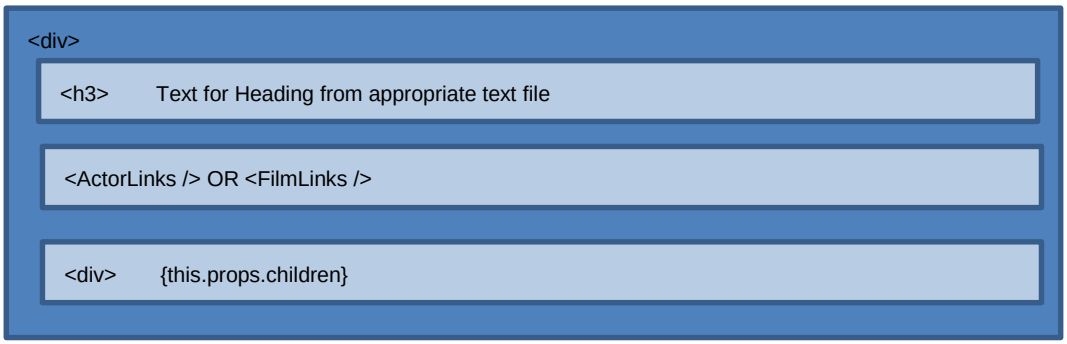


Figure 6 - Composition of Actors and Films Components

9.1. Create the `<Actors />` and `<Films />` components using the appropriate Heading text from the text file and the appropriate ‘Links’ component, remembering to `import` them.

The <ActorLinks /> and <FilmLinks /> components are almost identical too. The differences in them are obvious! They are laid out as shown:

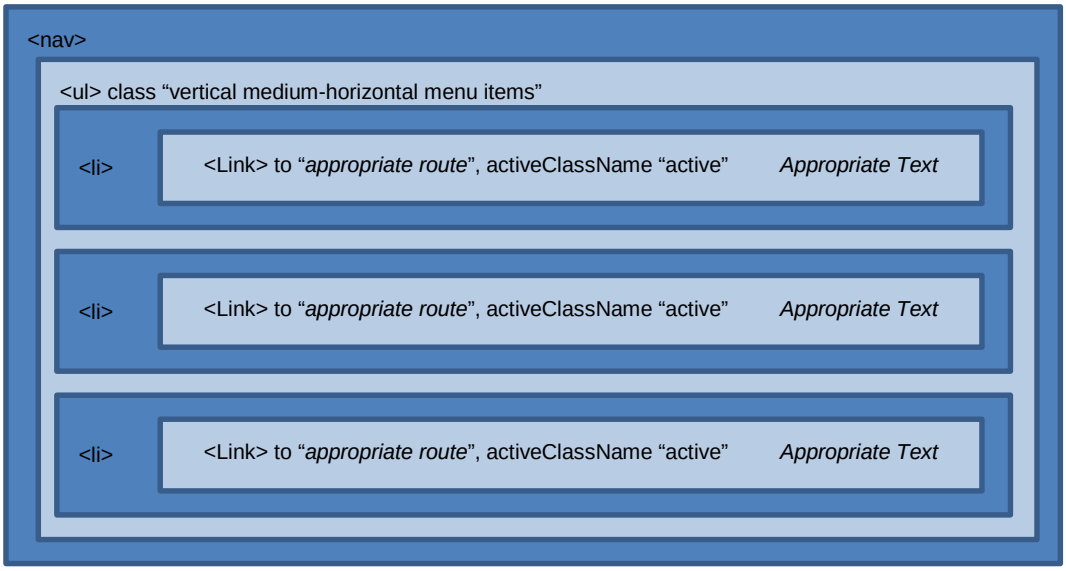


Figure 7 - Composition of 'Links' Components

9.2. Create the two ‘Links’ components. Use the appropriate subroute address for the `to` attribute of each link and appropriate text for the link.
- *Use* `/actors/FirstNameLastName` *for actors* `and /films/filmID` *for films*

Clicking on a link, either in Films or Actors, needs to render a component to display the requested information. This will be done in the `<ActorsDetails />` and the `<FilmDescription />` components. These will serve as the children of the respective parent components and will be displayed through the `{this.props.children}` call and are referenced in the route set out in **main.js**.

Whilst the actual rendered component is simple, there is quite a bit of JavaScript that needs to execute to allow the correct information to be displayed!

This guide will explain how ActorDetails is implemented, allowing you to reproduce code to implement the FilmDescription without further guidance.

9.3. Open or create a **ActorDetails.jsx** file in the **actors** folder.

9.4. Add the import statement for React.

9.5. Copy the `actors` array from the **actors.txt** file and include it as a global variable in the file.

9.6. Declare the ActorDetails class and add a constructor that takes an argument of props, calls super and sets initial state as:

```
      actor_name : actors[0].actor_name,

     character_name : actors[0].character_name,
```

```
    bio : actors[0].bio
```

9.7. Define a function called `getActorIndex` to retrieve the index in the array that an actor's name sits at. The function should:

- Take an argument of `actorName`
- Declare a block level variable of `actorIndex`, initialised to zero
- Loop through the `actors` array
    - Check to see if the `actor_name` at the index of the value of the loop counter is the same as the `actorName` supplied to the function
        - *HINT: Use the `replace` function with the argument `(/\s+g, '')` to remove the space in the `actor_name` from the array.*
    - If a match is found, set `actorIndex` to the current loop counter value and then `break` from the loop
- `return` the value of `actorIndex`.

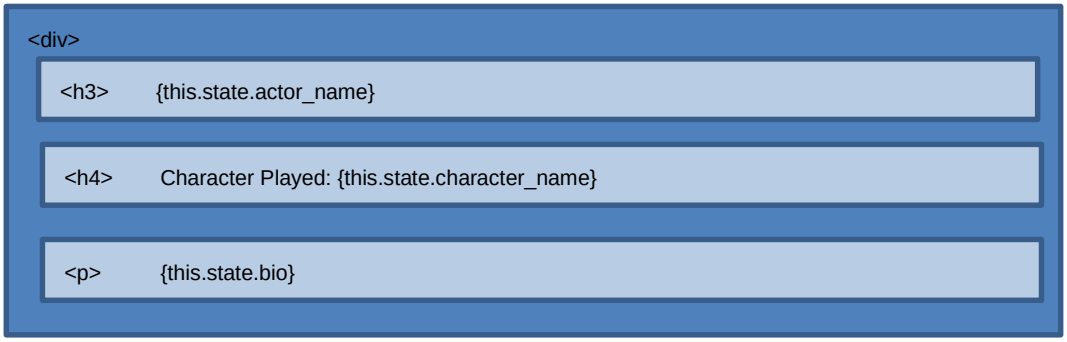
9.8. Define a function called `setActorDetails` to change the state of the component. It should take an argument of `index` and:

- Use `this.setState()` to set values to those stored in the `actors` array at the supplied `index` value

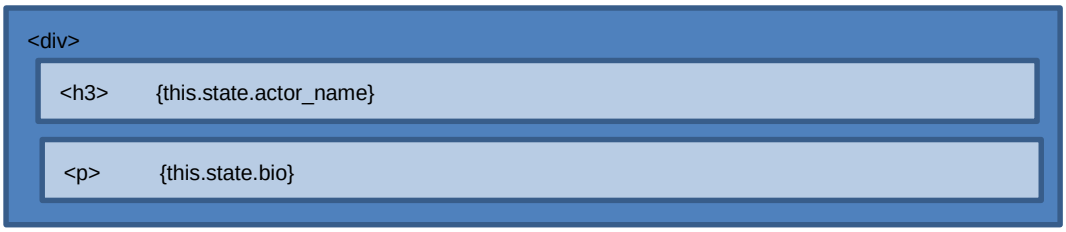
9.9. To affect the change of the rendered component, the lifecycle method `componentWillReceiveProps` will be used. It will take the argument of `nextProps` and should:

- Define a block level variable called `name` that is set to `nextProps.params.actorName`
    - This essentially gets the parameter that is used in the route when a link is clicked
- Define a block level variable called `index` and set this to be the return of the `getActorIndex` function (don't forget to use `this`!)
- Call the `setActorDetails` method, passing in the value of `index` (don't forget to use `this`!)

9.10. Finally, define the render method to return the structure shown below:



The <FilmDescription /> component is almost identical to the <ActorDetails /> component in that it needs an array, a method to get the index matched in the array, a method to set the state and a method to call these methods from the lifecycle method. The render method is similar too, but should return the following structure:



Have a go at creating the <FilmDescription /> component. If you get stuck, call your instructor.

Once you have completed the application, save all files and view it. If you need any help debugging, again, call your instructor.

Appendix

This application could have been created without using the subroutes. An implementation of this can be found in the solution2 folder.

Foundation documentation:

http://foundation.zurb.com/frameworks-docs.html

jquery documentation:

http://api.jquery.com/

what-input documentation:

https://github.com/ten1seven/what-input