|  |  |
| --- | --- |
| **Exercise** | React.1 |
| **App** | (None – create a new **React** app from scratch) |
| **Purpose** | Get up-and-running with **React** |
| **Description** | We will usually use the tool **create-react-app** to create a “scaffolding” for a **React** app for us. Once created, we can adapt the scaffolding to our specific needs. |
| **Steps** | 1. Start up Visual Studio Code. 2. Open a terminal window, and run: **> npm install –g create-react-app**. This will install the **create-react-app** tool, and make it globally available. **NB**: it may take several minutes before the installation is complete. 3. Now open a terminal window in the root of the folder where you wish to create **React** apps (you could e.g. call the folder **ReactApps**, but it is up to you), and run **> create-react-app helloworld**. This will create a new **React** app for you, in the folder **helloworld**. **NB**: it may (again) take several minu­tes before the creation process is complete. 4. Once the creation is done, open a terminal window in the app folder, and run **> npm start**. This will start up the **React** development server (you can shut down the server using **Ctrl+C** later on), and your app will show up in your browser at **localhost:3000**. This may also take a few seconds… 5. Now delete the content – but not the folder itself – of the **src** folder, and create a new file **index.js** in that folder. **NB**: It must have this exact name! 6. Copy-paste the below content into the index.js file, and verify that the browser now shows **“Hello World!”** at **localhost:3000**.   //Import the React and ReactDom libs  import React from 'react';  import ReactDOM from 'react-dom';  //Create a React component named "App"  const App = () => {  return (  <div>  <h1>Hello world!</h1>  </div>  );  };  //Take the "App" component and show it on the screen  ReactDOM.render(<App/>, document.querySelector('#root')); |

|  |  |
| --- | --- |
| **Exercise** | React.2 |
| **App** | (None – create a new **React** app from scratch) |
| **Purpose** | Implement a fairly simple **React** app, for presentation of a fixed number of data instances. |
| **Description** | In the **Data** folder, you will find four image files and an **info.docx**, which contains data describing four “roles” from a simple role-play game. |
| **Steps** | 1. Create a brand new **React** app from scratch, using **create-react-app** 2. Implement the app, such that it presents the given data in a reasonable manner (it’s up to you to define “reasonable” ☺). You can e.g. use the provi­ded examples as a guide for your implementation, along with the **React.pptx** powerpoint presentation. |

|  |  |
| --- | --- |
| **Exercise** | React.3 |
| **App** | (None – create a new **React** app from scratch) |
| **Purpose** | Implement a more complex **React** application, including calls to a Web API, and presentation of variable-sized data. |
| **Description** | A typical Single-Page app (SPA) will include retrieval of data from a Web API, and presentation of this data – which may e.g. involve a variable number of data instances – in the UI.  The main focus of this exercise is to get such an application up-and-running, but perhaps with limited/no user interaction. The application can then be extended with a fuller UI next week. |
| **Steps** | 1. Create a brand new React app from scratch, using **create-react-app** 2. Implement the app; the exact “theme” for the app is up to you, but a good and relevant example could – as always – be an app which make use of the **MovieDB** Web API. However, feel free to choose something else. The main goals for the application should be to make use of:    1. Multiple **React** components (class and/or functional)    2. The **props** system    3. The **state** system    4. Callbacks (perhaps)    5. Web API calls (e.g. using **axios**)    6. **filter**/**map** methods    7. Styling (with Bootstrap or some other library) |

|  |  |
| --- | --- |
| **Exercise** | React.4 |
| **App** | **color-redux** (**NB**: Note that this is a **redux**-only project, so you can use the project as-is. You do NOT need to run **npm install** first. |
| **Purpose** | Implement a simple App which uses **Redux** for state management, but without involving **React**. |
| **Description** | The App is intended to be very similar to the example **inc-redux**, so you can solve the exercise by following the same steps as outlined in that example. |
| **Steps** | 1. Open **index.html**, and take note of the **<a>** tag with **id = value**, and the three buttons beneath it. Each button makes a call to **store.dispatch**, but the calls will not work properly until you have completed the implemen­tation of the code in **index.js** 2. Open **index.js**, and follow the steps described in the code. Again, remem­ber that you can use the corresponding **index.js** file from the **inc-redux** example as a guideline. 3. Keep working until you can change the color of the smiley face by clicking on the three buttons below it. |

|  |  |
| --- | --- |
| **Exercise** | React.5 |
| **App** | **color-reactredux** |
| **Purpose** | Re-implement the application from the previous exercise, but this time as a **React/Redux** application. Use standard folder and file struc­ture for **React/Redux** applications. |
| **Description** | The App is intended to be functionally identical to the **color-redux** example from the previous exercise, but now implemented as a **React** app using **Redux** for state management. Note that the **src** folder for the given application now contains a number of new subfolders:   * actions * components * reducers   These folders reflect the “standard” for how to organise files in a **React/Redux** project. The folders also contain the standard files as well, but they are initially empty. |
| **Steps** | 1. Get the application from **GitHub**, open it in Visual Studio Code, open a terminal window in the root folder of the App, and run the **npm install** command without any parameters. 2. Once the installation is complete, you can run **npm start**. Initially, nothing will show up in the browser. 3. Implement the same functionality as seen in the **color-redux** exam­ple, but by using the **React/Redux** approach. You can use the **inc-reactredux** app as a guideline, since that example has the same file structure. If you feel insecure about how to get started on the implementation, take some time to study the **inc-reactredux** example once more. |

|  |  |
| --- | --- |
| **Exercise** | React.6 |
| **App** | (None – create a new **React** app from scratch, and also install the **redux** and **react-redux** packages) |
| **Purpose** | Implement a more complex application using **React** in general, and using **Redux** for state management. |
| **Description** | The goal of this application should be to use **Redux** for state management, in a setup where the application state is a bit more complex. A couple of examples could be:   * A very simple role-play game, where two opponents can attack each other. Each player should have a number of hit points, which will decrease with a certain amount (fixed or randomised) when attacked. A “battle log” could be included, which lists the progression of the battle. * A dice-like game, where you can roll one or more dice, and perhaps add the value of the roll to a sum. * A simulation of a simple shop, e.g. an extension of the **lemonadestand-reactredux** example, with a slightly more complex inventory system (or maybe a sales history functionality).   These are just examples; feel free to choose a different theme for your applica­tion. Just remember to try to challenge yourself a bit! |
| **Steps** | 1. **[mandatory]** Implement the application. 2. **[Optional]** Demo your application for the class next week ☺ |

|  |  |
| --- | --- |
| **Exercise** | React.7 |
| **App** | None – create a new **React** app with **create-react-app**, and also install these packages: **redux,** **react-redux**, **axios** and **redux-thunk** |
| **Purpose** | Implement an application which uses a Web API for retrieving data. |
| **Description** | The example **f1-react** in the **Apps** folder uses an external Web API to retrieve data on Formula 1 drivers for the 2018 season. If we are using **Redux** for state manage­ment, we do however have to handle the asynchronous call to the Web API in a special man­ner, since the standard **Redux** model for invoking the **dispatch** func­tion isn’t comtatible with asynchronous calls. In order to manage this, we have to install a small piece of “middleware”, specifically the **redux-thunk** package.  In the app **f1-reactreduxthunk** (also found in the **Apps** folder), you can see an exam­ple of how to invoke a Web API correctly, using the **redux-thunk** middle­ware package and **axios** for actually executing the API call. |
| **Steps** | 1. Make sure you understand how the **f1-reactreduxthunk** app works. Pay particular attention to the below:    1. [in **/src/index.js**] The way the **thunk** middleware is injected into the **Redux** store.    2. [in **/src/actions/index.js**] How we execute the Web API call, and how the response from the call is used to call **dispatch** explicitly with an action object.    3. [in **/src/components/DriverList.js**] How we implement this com­po­nent as a class-based **React** component, and make the call to the action creator **fetchF1Data** (which in turn invokes the Web API call) in the life-cycle method **componentDidMount**. 2. Once you feel you understand the app, implement a similar app yourself. That is, the app should as a minimum contain a call to a Web API, and should somehow display the result of the call. If you cannot find a suitable Web API to call, you can use one of the the **JSONPlaceholder** Web APIs (<https://jsonplaceholder.typicode.com> ). An example of a Web API call is: (<https://jsonplaceholder.typicode.com/users> ), which return JSON data. |

|  |  |
| --- | --- |
| **Exercise** | React.8 |
| **App** | None – create a new **React** app with **create-react-app**, and also install the package **react-router-dom** (**NB**: this does not involve **Redux**!) |
| **Purpose** | Implement an application which uses **React** routing. |
| **Description** | The example **club-reactrouter** uses **React** routing for enabling navigation in the app. The main idea in **React** routing is that a number of routing-related **React** components become available after installing the **react-router-dom** package. These components are then used to manage internal routing in the app. |
| **Steps** | 1. Make sure you understand how the **club-reactrouter** app works. Pay particular attention to the below:    1. [in **/src/index.js**] At the top level of the app, we wrap the **<App>** component into a **<BrowserRouter>** component.    2. [in **/src/components/App.js**] The **<App>** component itself is divi­ded into a **<Header>** and a **<Main>** component.    3. [in **/src/components/Header.js**] The **<Header>** component con­tains **<Link>** components, one for each link to a “subcomponent”. Note the **to** attrbute, which specifies the sub-URL we are trying to navigate to.    4. [in **/src/components/Main.js**] This component primarily contains three **<Route>** components, each specifying a **path** attribute. This is the attribute which will be matched with the **to** attribute from any **<Link>** component. The **<Switch>** component ensures that only the first **<Link>** component matching the path is selected.    5. Also take a look at how routing is used in **<Roster>**, **<FullRoster>** and **<Player>**. 2. Once you feel you understand the app, implement a similar app yourself. You can choose the theme for your app freely. It could be a presentation of yourself, a hobby, or whatever you feel could be fun ☺. |