Managing State in React.js Application

# 1 Introduction

In this assignment we are going to practice working with application and component level state. ***State*** is the collection of data values stored in the various constants, variables and datastructures in an application. ***Application state*** is data that is relevant across the entire application or a significant subset of related components. ***Component state*** is data that is only relevant to a specific component or a small set of related components. If information is relevant across several or most components, then it should live in the application state. If information is relevant only in one component, or a small set of related components, then it should live in the component state. For instance, the information about the currently logged in use could be stored in a profile, e.g., ***username***, ***first name***, ***last name***, ***role***, ***logged in***, etc., and it might be relevant across the application. On the other hand, filling out shipping information might only be relevant while checking out, but not relevant anywhere else, so shipping information might best be stored in the ***ShippingScreen*** or ***Checkout*** components in the component's state. We will be using [the ***Redux*** state management library](https://redux.js.org/) to handle application state, and use ***React.js*** state and effect hooks to manage component state.

# 2 Labs

This section presents ***React.js*** examples to program the browser, interact with the user, and generate dynamic HTML. Use the same project you worked on last assignment. After you work through the examples you will apply the skills while creating a ***Tuiter*** on your own. Using ***IntelliJ***, open the project you created in previous assignments. From within IntelliJ, use ***File***, ***Open Project***, and navigate to the project directory, (***tuiter-react-web-app***), and click ***Open*** or ***OK***. ***Include all the work in the Labs section as part of your final deliverable***. Do all your work in a new branch called ***a7*** and deploy it to Netlify to a branch deployment of the same name. TAs will grade the final result of having completed the whole ***Labs*** section.

## 2.1 Installing Redux

As mentioned earlier we will be using [the ***Redux*** state management library](https://redux.js.org/) to handle application state. To install ***redux***, type the following at the command line from the root folder of your application.

| **$** **npm install redux** |
| --- |

After redux has installed, install ***react-redux*** and the redux ***toolkit***, the libraries that integrate ***redux*** with ***React.js***. At the command line, type the following commands.

| **$** **npm install react-redux**  **$** **npm install @reduxjs/toolkit** |
| --- |

## 2.2 Create an Assignment 7 component

To get started, create an assignment 7 component that will host all the exercises in this assignment. Then import the component into the labs component created in an earlier assignment. While you're in the labs component, add routes so that each assignment will appear in its own screen when you navigate to labs and then to ***/a7***. That is, make the assignment 6 component the default element that renders when navigating to the labs path and map assignment 7 to the ***/a7*** path. You'll need to change the lab component route in ***App.js*** so that all routes after ***/\**** are handled by the routes declared in the labs component, e.g., <**Route path="/\*" element=**{<**Labs**/>}/>. Use the code snippets below as a guide.

| ***src/labs/a7/index.js*** | ***src/labs/index.js*** | ***src/nav.js*** |
| --- | --- | --- |
| **import *React* from "react"**;  **const** *Assignment7* = () => {  **return**(  <>  <**h1**>Assignment 7</**h1**>  </>  );  };  **export default** *Assignment7*; | **import** *Nav* **from "../nav"**;  **import** *Assignment6* **from "./a6"**;  **import** *Assignment7* **from "./a7"**;  **import** {*Routes*, *Route*} **from "react-router"**;  **function** *Labs*() {  **return** (  <**div**>  <**Nav**/>  <**Routes**>  <**Route index**  **element=**{<**Assignment6**/>}/>  <**Route path="a7"**  **element=**{<**Assignment7**/>}/>  </**Routes**>  </**div**>  );  }  **export default** *Labs*; | **import** {*Link*} **from "react-router-dom"**;  **function** *Nav*() {  **return** (  <**div**>  <**Link to="/"**>Labs</**Link**> |  <**Link to="/"**>Assignment 6</**Link**> |  <**Link to="/a7"**>Assignment 7</**Link**> |  <**Link to="/tuiter"**>Tuiter</**Link**>  </**div**>  );  }  **export default** *Nav*; |
|  |  |  |

## 2.3 Create a Redux Examples component

To learn about redux, let's create a redux examples component that will contain several simple redux examples. Create an ***index.js*** file under ***src/labs/a7/redux-examples/index.js*** as shown below. Import the new redux examples component into the assignment 7 component so we can see how it renders as we add new examples. Reload the browser and confirm the new component renders as expected.

| ***redux-examples/index.js*** | | ***a7/index.js*** |
| --- | --- | --- |
| **import *React* from "react"**;  **const** *ReduxExamples* = () => {  **return**(  <**div**>  <**h2**>Redux Examples</**h2**>  </**div**>  );  };  **export default** *ReduxExamples*; | | **import *React* from "react"**;  **import** *ReduxExamples* **from "./redux-examples"**;  **const** *Assignment7* = () => {  **return**(  <>  <**h1**>Assignment 7</**h1**>  <**ReduxExamples**/>  </>  );  };  **export default** *Assignment7*; |

## 2.4 Create a Hello World Redux component

Our first example will be the simplest redux example. Create a reducer that provides some static data, e.g., a hello message. Copy the code below into ***src/labs/a7/redux-examples/reducers/hello.js***. Notice that we could have stored the data ***{message: 'Hello World'}*** into a static JSON file, but the point here is to learn how data can be shared across multiple components, and how each can interact with the data like reading and writing to it. To do that we wrap the data in a function that can calculate the data dynamically as circumstances change over time.

| ***hello.js*** | |
| --- | --- |
| **const** *hello* = () => ({**message**: **'Hello World'**});  **export default** *hello*; | |

Now let's create a component that can retrieve the data from the reducer and display it in a React.js component. In ***src/labs/a7/redux-examples/hello-redux-example-component.js***, copy the code shown below. The component uses redux's ***useSelector*** hook to extract the message from the reducer. When the component loads, reducers pass their data in the function declared in ***useSelector***. In the code below, the parameter ***hello*** in ***(hello) => { … }***, gets the object returned by the reducers, e.g., ***{message: 'Hello World'}***, therefore, ***(hello) => hello.message*** returns 'Hello World', and that's the value const message is initialized with. The component goes on to render ***'Hello World'*** in an ***H1*** element.

| ***hello-redux-example-component.js*** |  |
| --- | --- |
| **import *React* from "react"**;  **import** {*useSelector*} **from "react-redux"**;  **const** *HelloReduxExampleComponent* = () => {  **const** message = *useSelector*((hello) => hello.message);  **return**(  <**h3**>{message}</**h3**>  );  };  **export default** *HelloReduxExampleComponent*; | *// import useSelector hook // from react-redux*  *// extract 'Hello World' from reducer*  *// render <h1>Hello World</h1>* |

Now we have to glue together the reducer producing the data, and the ***HelloReduxExampleComponent*** consuming the data. We connect the two -- data source and data consumer -- through a ***Provider*** as shown below in ***redux-examples/index.js***.

| ***redux-examples/index.js*** | |
| --- | --- |
| **import *React* from "react"**;  **import** *HelloReduxExampleComponent*  **from "./hello-redux-example-component"**;  **import** *hello* **from "./reducers/hello"**;  **import** {***createStore***} **from "redux"**;  **import** {Provider} **from "react-redux"**;  **const** store = ***createStore***(*hello*);  **const** *ReduxExamples* = () => {  **return**(  <**Provider store=**{store}>  <**div**>  <**h2**>Redux Examples</**h2**>  <**HelloReduxExampleComponent**/>  </**div**>  </**Provider**>  );  };  **export default** *ReduxExamples*; | *// import the component that consumes the data*  *// import reducer that calculates/generates the data*  *// import createStore to store data from reducers*  *// import Provider which will deliver the data*  *// create data storage*  *// Provider delivers data in store to child elements,*  *// component consumes the data* |

Refresh the browser and confirm that the ***HelloReduxExampleComponent*** renders the message from the reducer.

## 2.5 Retrieving state from a reducer

Redux allows maintaining the state of an application. The state changes over time as the user interacts with the application. There are four basic ways we interact with data: create data, read data, update date, and delete data. We often refer to these operations by the acronym CRUD. Let's implement a small todo app to illustrate the CRUD operations. In the same ***reducers*** directory created earlier, create the reducer for the todo app in a file called ***todos-reducer.js***. Copy the content below into the file.

| ***todos-reducer.js*** | |
| --- | --- |
| **const** initialTodos = [  {  **\_id**: **"123"**,  **do**: **"Accelerate the world's transition to sustainable energy"**,  **done**: **false**  },  {  **\_id**: **"234"**,  **do**: **"Reduce space transportation costs to become a spacefaring civilization"**,  **done**: **false**  },  ];  **const** todosSlice = *createSlice*({  **name**: **'todos'**,  initialState: initialTodos,  });  **export default** todosSlice.**reducer** | |

Notice that the ***todos-reducer.js*** declares an initial set of todo objects in a constant array. This will be the initial state of our simple todos application. We will then practice how to mutate the state in later lab exercises. All reducers must collate their collective states into a common ***store***. To do this we will use ***configureStore*** to collate the various reducers into a single store as shown below. In ***redux-examples/index.js***, import the new ***todos*** reducer and combine it with the existing ***hello*** reducer.

| ***redux-examples/index.js*** |  |
| --- | --- |
| **import *React* from "react"**;  **import** *HelloReduxExampleComponent*  **from "./hello-redux-example-component"**;  **import** *hello* **from "./reducers/hello"**;  **import** *todos* **from "./reducers/todos-reducer"**;  **import** {Provider} **from "react-redux"**;  **~~import~~** ~~{~~***~~createStore~~***~~}~~ **~~from "redux"~~**~~;~~  **import** { *configureStore* }  **from '@reduxjs/toolkit'**;  **import** *Todos* **from "./todos-component"**;  **const** store = *configureStore*({  **reducer**: {*hello*, *todos*}  });  **const** *ReduxExamples* = () => {  **return**(  <**Provider store=**{store}>  <**div**>  <**h2**>Redux Examples</**h2**>  <**Todos**/>  <**HelloReduxExampleComponent**/>  </**div**>  </**Provider**>  );  };  **export default** *ReduxExamples*; | *// import the new reducer*  *// instead of createStore,*  *// import the* ***configureStore*** *function*  *// import new component to render todos*  *// combine all* ***reducers*** *into a single store*  *// each available through these namespaces*  *// render todos component (see below)* |

The ***Provider*** delivers the content of the ***store*** to all its child components. This is done by invoking all the methods declared in ***useSelector*** in the components. Copy the code snippet below in a new file ***redux-examples/todos-component.js***. The component uses ***useSelector*** to retrieve the todos generated by ***todos-reducer.js***. The ***todos*** is retrieved from the reducer with ***useSelector*** returning the ***todos*** arrays returned by the reducer, e.g., the array of two todo objects in ***todos-reducer.js***.

| ***todos-component.js*** |  |
| --- | --- |
| **import *React* from "react"**;  **import** {*useSelector*} **from "react-redux"**;  **const** *Todos* = () => {  **const** todos  = *useSelector*(state => state.todos);  **return**(  <>  <**h3**>Todos</**h3**>  <**ul className="list-group"**>  {  todos.map(todo =>  <**li className="list-group-item"**>  {todo.**do**}  </**li**>  )  }  </**ul**>  </>  );  };  **export default** *Todos*; | *// import* ***useSelector***  *// retrieve* ***todos*** *from reducer state and assign to*  *// local* ***todos*** *constant*  *// iterate over* ***todos*** *array and render a*  *// line item element for each* ***todo*** *object*  *// display* ***do*** *property containing the todo text* |

Before we implemented the ***todos-reducer***, we only had the ***hello*** reducer. When we combined the reducers we bound them to attributes ***hello*** and ***todos***: **const** store = *configureStore*({**reducer**: {*hello*, todos}}). The state of each reducer is now accessible through these properties. We now need to retrieve the message from the ***hello*** sub state as shown below.

| ***hello-redux-example-component.js*** | |
| --- | --- |
| **const** *HelloReduxExampleComponent* = () => {  **const** message = *useSelector*((state) => state.hello.**message**);  **return**(  <**h1**>{message}</**h1**>  );  }; | |

## 2.6 Working with forms and local state

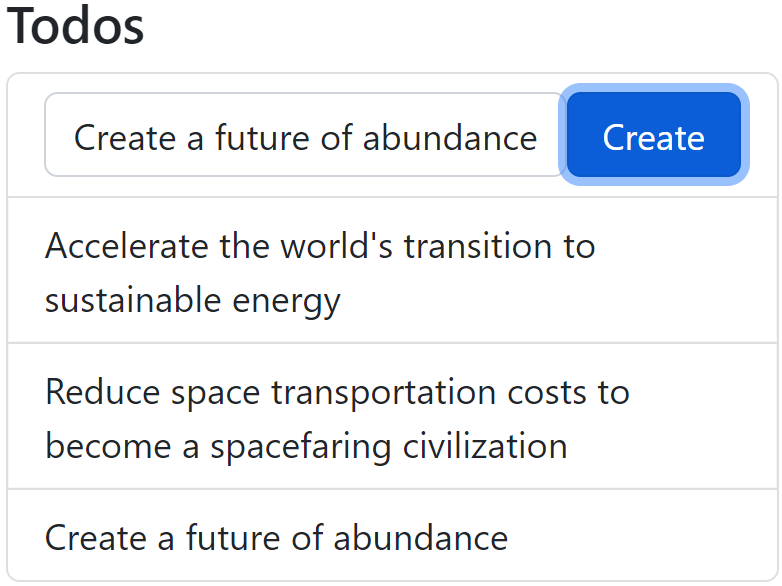
Redux is great for working with application level state. Let's now consider component state. The React ***useState*** hook can be used to deal with local component state. This is especially useful to integrate React with forms. Let's practice working with forms by adding an input field users can use to create new todos. We'll keep track of the new todo's text in a local state variable called ***todo*** and mutate its value using a function called ***setTodo*** as shown in the code below.

| ***todos-component.js*** |  |
| --- | --- |
| **import *React***, {*useState*} **from "react"**;  **import** {*useSelector*} **from "react-redux"**;  **const** *Todos* = () => {  **const** todos =  *useSelector*(state => state.todos);  **const** [todo, setTodo] = *useState*({**do**: **''**});  **const** todoChangeHandler = (event) => {  **const** doValue = event.**target**.**value**;  **const** newTodo = {  **do**: doValue  };  setTodo(newTodo);  }  **return**(  ...  <**ul className="list-group"**>  <**li className="list-group-item"**>  <**input**  **onChange=**{todoChangeHandler}  **value=**{todo.**do**}  **className="form-control"**/>  </**li**>  ...  );  };  **export default** *Todos*; | *// import useState to work with local state*  *// create todo local state variable*  *// handle keystroke changes in input field*  *// get data from input field*  *// create new todo object instance*  *// setting the todo's do property*  *// change local state todo variable*  *// add a new line item at the top*  *// containing an input field to type todo*  *// handle keystrokes to update component state*  *// update field with latest state value* |

## 2.7 Handling application level events

Now that we have edited a todo object, we can send it to the reducer to store it in the global state. Lets add an ***addTodo*** handler that can receive the new todo instance and push it to the array of current todos.

| ***todos-reducer.js*** | |  |
| --- | --- | --- |
| **const** todosSlice = *createSlice*({  **name**: **'todos'**,  initialState: initialTodos,  **reducers**: {  addTodo(state, action) {  state.push({  **\_id**: (**new *Date***()).getTime(),  **do**: action.**payload**.**do**,  **done**: **false**  });  },  }  });  **export const** {*addTodo*} = todosSlice.**actions**  **export default** todosSlice.**reducer** | | *// define reducer functions as a map*  *// reducer functions receive current state*  *// mutate current state into new state, e.g.,*  *// pushing new object. \_id set to current date*  *// do set to "do" object sent through action obj*  *// commonly referred to as the "payload"*  *// export actions so we can call them from UI* |



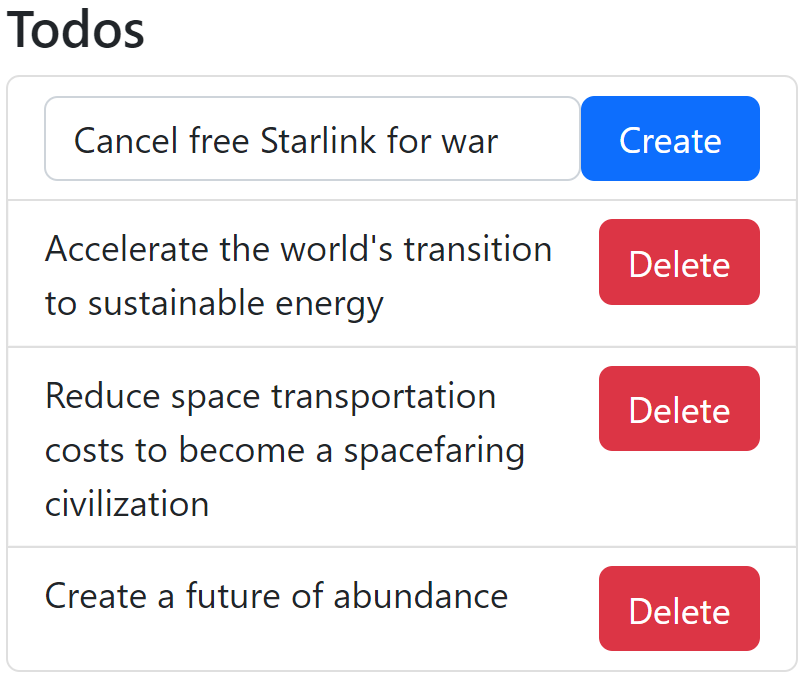
To send it to the reducer we use the ***useDispatch*** hook as shown below. A new button handles the click event invoking a new ***createTodoClickHander*** which dispatches the new ***todo*** through the ***addTodo*** function implemented in the todos reducer above. Reload the page and confirm you can add new todos.

| ***todos-component.js*** | |  |
| --- | --- | --- |
| **import *React***, {*useState*} **from "react"**;  **import** {*useDispatch*, *useSelector*}  **from "react-redux"**;  **import** {*addTodo*}  **from "./reducers/todos-reducer"**;  **const** *Todos* = () => {  **const** todos = *useSelector*(state => state.**todos**);  **const** [todo, setTodo] = *useState*({**do**: **''**});  **const** dispatch = *useDispatch*();  **const** createTodoClickHandler = () => {  dispatch(*addTodo*(todo))  }  **...**  **return**(  <>  <**h3**>Todos</**h3**>  <**ul className="list-group"**>  <**li className="list-group-item"**>  <**button onClick=**{createTodoClickHandler}  **className="btn btn-primary w-25**  **float-end"**>  Create</**button**>  <**input onChange=**{todoChangeHandler}  **value=**{todo.**do**}  **className="form-control w-75"**/>  ... | | *// useDispatch hook to call reducers*  *// import reducer function exported by*  *// todos-reducer*  *// get distacher to invoke reducer functions*  *// handle click event of button*  *// call reducer function passing new todo*  *// as the payload in the action object*  *// new button to add new todo*  *// calls function to handle click event* |

## 2.8 Deleting from application state

We can delete todos by splicing out the deleted todo from the current array of todos. To start let's add a delete button to all the todos and bind a click event handled by the event handler as shown below. The ***map()*** function takes two arguments, the first one being the element in the current iteration, and the second one the index of the element in the array. Let's pass this index to the event handler, and then pass it on to the ***deleteTodo*** reducer function as the payload.

| ***todos-component.js*** | |
| --- | --- |
| **import {*addTodo*, *deleteTodo*}**  **from "./reducers/todos-reducer";**  ...  **const** *Todos* = () => {  ...  **const** dispatch = *useDispatch*();  **const** deleteTodoClickHandler = (index) => {  dispatch(*deleteTodo*(index))  }  **const** createTodoClickHandler = () => {  dispatch(*addTodo*(todo))  }  ...  **return**(  ...  {  todos.map((todo, index) =>  <**li key=**{todo.**\_id**}  **className="list-group-item"**>  <**button onClick=**{() =>  deleteTodoClickHandler(index)}  **className="btn btn-danger**  **float-end ms-2"**>  Delete  </**button**>  {todo.**do**}  </**li**>  )  }  ...  );  };  **export default** *Todos*; | *// import a new deleteTodo reducer function*  *// implemented below*  *// handle delete button click, accepts todo index*  *// dispatch event to deleteTodo reducer function*  *// passing index of todo we want to delete*  *// add index parameter*  *// new Delete button sends index of todo to*  *// delete to handler. Note () => {} because*  *// we are passing index parameter otherwise*  *// gets into infinite loop* |



The dispatch will send the index of the delete object we want to remove as the payload of the action object. Implement a new ***deleteTodo*** reducer function as shown below that accepts the index in the action's payload and then uses it to splice out the todo object from the state's todo array. Refresh the Website and confirm that you can add new todos and then delete them.

| ***todos-reducer.js*** | |
| --- | --- |
| ...  **reducers**: {  addTodo(state, action) {  state.push({  **\_id**: (**new *Date***()).getTime(),  **do**: action.**payload**.**do**,  **done**: **false**  });  },  deleteTodo(state, action) {  **const** index = action.**payload**  state.splice(index, 1)  },  }  ...  **export const** {*addTodo*, *deleteTodo*} = todosSlice.**actions** | *// new deleteTodo function extracts*  *// index from action's payload and*  *// uses it to splice out the todo to*  *// be deleted* |

## 2.9 Updating application state

Let's practice changing something in a reducer. To do this, let's add a ***done*** flag we can toggle with a checkbox. Add a checkbox at the beginning of the todo that is checked if the todo's done field is true and unchecked otherwise. If a user checks the button, we'll pass the ID of the corresponding todo object to a handler that will pass the ID to a reducer function. Use the code snippet below as a guidance.

| ***todos-component.js*** | |
| --- | --- |
| **import** {*addTodo*, *deleteTodo*, todoDoneToggle}  **from "./reducers/todos-reducer"**;  **const** *Todos* = () => {  ...  **const** dispatch = *useDispatch*();  **const** toggleTodoDone = (todo) => {  dispatch(*todoDoneToggle*(todo))  }  **return**(  <>  <**h3**>Todos</**h3**>  {  todos.map((todo, ndx) =>  <**li key=**{todo.**\_id**}  **className="list-group-item"**>  <**button onClick=**{() =>  deleteTodoClickHandler(ndx)}  **className="btn btn-danger**  **float-end ms-2"**>  Delete  </**button**>  <**input type="checkbox"**  **checked=**{todo.**done**}  **onChange=**{() =>  toggleTodoDone(todo)}  **className="me-2"**/>  {todo.**do**}  </**li**>  )  } | *// import reducer function implemented below*  *// handle checkbox click event, accept todo index*  *// send index to reducer function passing index*  *// of todo to toggle done field*  *// new checkbox which is checked*  *// if todo.done is true*  *// if user changes checkbox, we'll pass the*  *// todo to reducer function to update todo's*  *// state* |

In the todos reducer, add a ***todoDoneToggle*** reducer function that will find the todo by its ID, and then update its done field as shown below.

| ***todos-reducer.js*** | |
| --- | --- |
| ...  deleteTodo(state, action) {  **const** index = action.**payload**  state.splice(index, 1)  },  todoDoneToggle(state, action) {  **const** todo = state.find((todo) =>  todo.**\_id** === action.**payload.\_id**)  todo.**done** = !todo.**done**  }  }  });  **export const** {  *addTodo*,  *deleteTodo*,  *todoDoneToggle*  } = todosSlice.**actions** | |

# 3 Tuiter

Now that you've had a chance to practice with ***redux***, let's apply these skills on our ***Tuiter*** application from previous assignments. Do all your work in directory ***src/tuiter***.

## 3.1 Re-implement the WhoToFollowList component to use Redux

Ok, we're all setup to start adding ***redux*** to our ***Tuiter*** application. Let's start with the simplest of component: ***WhoToFollowList***. This should be easy since the component is read only and its state does not change, so it should be easy to port its current implementation so it uses redux instead of a static array. The original implementation used a JSON file ***who.json***. We'll refactor this to use a reducer that provides the same data as the JSON file. Let's put all tuiter related reducers into a new ***src/tuiter/reducers*** folder and move ***who.json*** into a new ***data*** folder in ***src/tuiter/data/who.json***. In the ***reducers*** folder create a reducer called ***who-reducer.js*** that just returns the data in ***who.json*** as shown below.

| ***who-reducer.js*** |  |
| --- | --- |
| **import** { *createSlice* } **from "@reduxjs/toolkit"**;  **import** whoArray **from "./who.json"**;  **const whoSlice = *createSlice*({**  **name: "who",**  **initialState: whoArray**  **});**  **export default whoSlice.reducer;** | *// import slice*  *// import data from JSON file*  *// create the slice*  *// name the reducer*  *// initialize the reducer's state*  *// export the reducer* |

Now let's add the state created by our new reducer and put it in a store so that we can then provide it to the application. In ***src/tuiter/index.js*** create store and provider as shown below. Your folder paths might differ.

| ***tuiter/index.js*** | |
| --- | --- |
| **import *React* from "react"**;  **import** *ExploreComponent* **from "./explore"**;  **import** *NavigationSidebar*  **from "./navigation-sidebar"**;  **import** *WhoToFollowList*  **from "./who-to-follow-list"**;  **import** {*Routes*, *Route*} **from "react-router"**;  **import** whoReducer  **from "./reducers/who-reducer"**;  **import** { *configureStore* }  **from '@reduxjs/toolkit'**;  **import** {*Provider*} **from "react-redux"**;  **const** store = *configureStore*(  {**reducer**: {**who**: whoReducer}});  **function** *Tuiter*() {  **return** (  <**Provider store=**{store}>  ...  </**Provider**>  )  }  **export default** *Tuiter* | *// import the reducer*  *// import configureStore*  *// import the Provider component*  *// configure the store*  *// provide the store to the rest*  *// of the application so it can*  *// pull from the global state* |

Once the state is in the store, any component in the body of the ***Provider*** can retrieve state from the store. In ***who-to-follow-list/index.js*** use the ***useSelector*** hook to retrieve the ***who*** slice from the ***state*** in the store.

| ***who-to-follow-list/index.js*** | |
| --- | --- |
| **import** *WhoToFollowListItem*  **from "./who-to-follow-list-item"**;  **~~import~~** ~~whoArray~~ **~~from "./who.json"~~**~~;~~  **import** {*useSelector*} **from "react-redux"**;  **const** *WhoToFollowList* = () => {  **const** whoArray = *useSelector*(  (state) => state.who);  **return**(  <**div**>  <**h1**>Who To Follow!!</**h1**>  </**div**>  );  };  **export default** *WhoToFollowList*; | *// we moved the data into the reducer instead*  *// import hook to retrieve state from reducer*  *// retrieve state from the store* |

## 3.2 Re-implement the PostSummaryList component to use Redux

The ***PostSummaryList*** component implemented in earlier assignments imported ***posts.json*** and iterated over the array rendering a ***PostSummaryItem*** instance for each post object in the array. This is fine if the array is static, but it's insufficient if we need to create new tuits, delete old ones, and edit existing tuits. To do that we are going to have to be able to mutate the state of the current list of tuits as we generate different events based on user input. First step will be to create a reducer that holds the state of all the posts. Let's create a tuits reducer that holds the tuits as its state and provides reducers to retrieve, create, update, and delete tuits. Under ***src/tuiter/tuits***, create a tuits reducer as shown below and copy posts.json into ***tuits/tuits.json***.

| ***tuits-reducer.js*** | |
| --- | --- |
| **import** { *createSlice* } **from "@reduxjs/toolkit"**;  **import** tuits **from './tuits.json'**;  **const** tuitsSlice = *createSlice*({  **name**: **'tuits'**,  **initialState**: tuits  });  **export default** tuitsSlice.**reducer**; |  |

In the tuiter component (***tuiter/index.js***) add the ***tuitsReducer*** to the store you already created earlier. Use the code below as a guide to combine reducers in ***tuiter/index.js***.

| ***tuiter/index.js*** | |
| --- | --- |
| **import *React* from "react"**;  **import** *ExploreComponent* **from "./explore"**;  **import** *NavigationSidebar*  **from "./navigation-sidebar"**;  **import** *WhoToFollowList* **from "./who-to-follow-list"**;  **import** {*Routes*, *Route*} **from "react-router"**;  **import** whoReducer  **from "./reducers/who-reducer"**;  **import** tuitsReducer **from "./tuits/tuits-reducer"**;  **import** { *configureStore* } **from '@reduxjs/toolkit'**;  **import** {*Provider*} **from "react-redux"**;  **const** store = *configureStore*({  **reducer**: {**who**: whoReducer, **tuits**: tuitsReducer}});  **function** *Tuiter*() {  ...  } | *// import the new tuits reducer*  *// add it to the store* |

Now the post summary list component can be re-implemented to use this new tuits reducer instead of the hard coded ***posts.json*** file. In the post summary list, replace the ***posts.json*** file with the new reducer as shown below. Refresh the screen and confirm that the tuits display as expected.

| ***posts-summary-list/index.js*** | |
| --- | --- |
| **import *React* from "react"**;  **import** *PostSummaryItem* **from "./post-summary-item"**;  **~~import~~** ~~postsArray~~ **~~from './posts.json'~~**~~;~~  **import** {*useSelector*} **from "react-redux"**;  **const** *PostSummaryList* = () => {  **const** postsArray = *useSelector*(state => state.**tuits**)  **return**(  <**ul className="list-group"**>  {  postsArray.map(post =>  <**PostSummaryItem**  **key=**{post.**\_id**}  **post=**{post}  />  )  }  </**ul**>  );  };  **export default** *PostSummaryList*; | *// replace getting tuits from a file*  *// to getting tuits from the store*  *// get tuits from the state in the store* |

## 3.3 Implement a Home component

Let's implement a new home component that will render when you click on the home link on the left navigation side bar. The component will consist of a list of tuits and a form to create new tuits. Let's first create a new ***TuitsList*** component that will render the list of tuits.

### 3.3.1 Implement tuits list component based on post summary

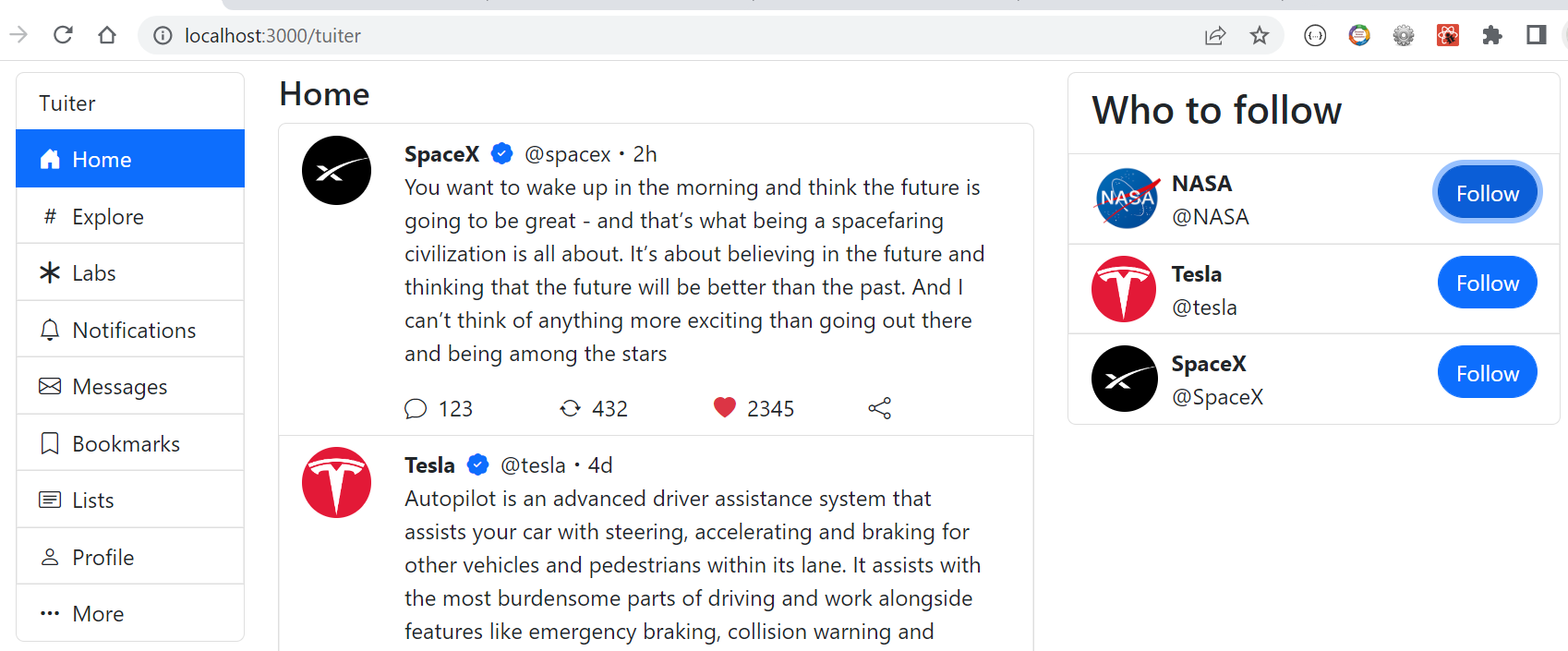
Using the ***PostSummaryItem*** and ***PostSummaryList*** components as an example, create new ***TuitItem*** and ***TuitsList*** components that render the same list of tuits, but with a slightly different layout as shown here on the right. Create the new components under ***src/tuiter/tuits*** and feel free to reuse the HTML and CSS you created for the home screen in earlier assignments. Implement the tuit statistics such as the replies, retuits, and likes in a separate component called ***TuitStats*** that renders a row of four columns as shown below each tuit here on the right. The first column displays the number of replies next to a message bubble icon. The second column displays the number of retuits next to a retuits icon. The third column displays the number of likes next to a heart icon. If the tuit has been liked, the heart should render in red or empty otherwise. If users like a tuit, the likes count increases by one and the heart turns red. If they unlike the tuit, the likes count decreases by one and the heart is not red. The last column displays a share icon. Feel free to use icons that you think better represent the data. To each of the tuit objects in ***src/tuits/tuits.json***, add new fields ***liked***, ***replies***, ***retuits***, ***likes***, ***handle***, and ***tuit*** with the following meaning.

* ***liked*** - is a boolean value (true/false) representing whether this tuit has been liked or not by the user
* ***replies*** - is an integer representing the number of times this tuit has been replied to
* ***retuits*** - is an integer representing the number of times this tuit has been retuited
* ***likes*** - is an integer representing the number of times this tuit has been liked
* ***handle*** - is a string that begins with @ representing the unique account name of the user who tuited
* ***tuit*** - is a string containing the main content of the tuit

Here's an example of the first tuit rendered above. Feel free to choose your own values for each of the tuit objects including the ***userName***, ***tuit***, ***liked***, ***likes***, ***retuits***, ***replies***, etc.

| ***src/tuits/tuits.json*** | |  |
| --- | --- | --- |
| [  {  **"\_id"**: 234,  **"topic"**: **"Space"**,  **"userName"**: **"SpaceX"**,  **"time"**: **"2h"**,  **"title"**: **"100s of SpaceX Starships land on Mars after a 6 month journey. 1000s of Martian colonists being building Mars Base 1"**,  **"image"**: **"spacex.png"**,  **"liked"**: **true**,  **"replies"**: 123,  **"retuits"**: 432,  **"likes"**: 2345,  **"handle"**: **"@spacex"**,  **"tuit"**: **"You want to wake up in the morning and think the future is going to be great - and that’s what being a spacefaring civilization is all about. It’s about believing in the future and thinking that the future will be better than the past. And I can’t think of anything more exciting than going out there and being among the stars"**  }, ...] | | |

Using the new tuits list component, create a new home component that imports and displays the new tuits list component as shown here below when you navitage to ***/tuiter***. The ***Home*** link in the navbar should be highlighted when the home screen displays.



## 

### 3.3.2 Creating a form

Let's learn how to work with forms using ***React*** and ***Redux***. To learn this new skill, let's create a textarea users can use to post new tuits as shown here on the right with the ***What's happening?*** placeholder. Implement the ***textarea***, icons and ***Tuit*** button as shown here, in a new file called ***whats-happening.js*** in the ***home*** directory. Feel free to use the code below as a guide. To interact with the textarea, implement a state variable called ***whatsHappening*** using the ***useState*** hook as shown below. Bind the value of the ***whatsHappening*** state variable to the ***textarea*** and its ***onChange*** event handler to update the state variable. Bind the ***Tuit*** button to a click event handler to confirm you have access to the ***whatsHappening*** state variable. In a later exercise you will send the state variable to the reducer to actually create the new tuit. The icons used here are bootstrap icons, but feel free to use your own icons.

| ***src/tuiter/home/whats-happening.js*** | |
| --- | --- |
| **import *React***, {*useState*} **from "react"**;  **const** *WhatsHappening* = () => {  **let** [whatsHappening, setWhatsHappening] = *useState*(**''**);  **const** tuitClickHandler = () => {  ***console***.log(whatsHappening);  }  **return** (  <**div className="row"**>  <**div className="col-auto"**>  <**img src="/images/nasa.png" width=**{60}/>  </**div**>  <**div className="col-10"**>  <**textarea value=**{whatsHappening} **placeholder="What's happening?"**  **className="form-control border-0"**  **onChange=**{(event) => setWhatsHappening(event.**target**.**value**)}>  </**textarea**>  <**div**>  <**button className="rounded-pill btn btn-primary float-end mt-2 ps-3 pe-3 fw-bold"**  **onClick=**{tuitClickHandler}>  Tuit  </**button**>  <**div className="text-primary fs-2"**>  <**i className="bi bi-card-image me-3"**></**i**>  <**i className="bi bi-filetype-gif me-3"**></**i**>  <**i className="bi bi-bar-chart me-3"**></**i**>  <**i className="bi bi-emoji-smile me-3"**></**i**>  <**i className="bi bi-geo-alt"**></**i**>  </**div**>  </**div**>  </**div**>  <**div className="col-12"**><**hr**/></**div**>  </**div**>  );  }  **export default** *WhatsHappening*; | |

Add the new ***WhatsHappening*** component to the top of the home screen in ***src/tuiter/home/index.js*** as shown below. Refresh the browser and confirm the ***HomeComponent*** renders the ***WhatsHappening*** and ***TuitsList*** components render as expected. Confirm you can type in the textarea.

| ***src/tuiter/home/index.js*** | |
| --- | --- |
| **import *React* from "react"**;  **import** *TuitsList* **from "../tuits/tuits-list"**;  **import** *WhatsHappening* **from "./whats-happening"**;  **const** *HomeComponent* = () => {  **return**(  <>  <**h4**>Home</**h4**>  <**WhatsHappening**/>  <**TuitsList**/>  </>  );  };  **export default** *HomeComponent*; |  |

### 3.3.3 Creating Tuits

Let's practice creating new tuits. In the reducer implemented in ***src/tuiter/tuits/tuits-reducer.js***, implement the ***createTuit*** reducer function that will add the new tuit to the array of tuits stored in the state. Add default attributes such as ***\_id***, ***topic***, ***userName***, ***handle***, etc as shown below. Copy the tuit in the action's payloadobject that includes the ***whatsHappening*** attribute sent by the dispatcher.

| ***tuits-reducer.js*** | |  |
| --- | --- | --- |
| **import** { *createSlice* } **from "@reduxjs/toolkit"**;  **import** tuits **from './tuits.json'**;  **const** currentUser = {  **"userName"**: **"NASA"**,  **"handle"**: **"@nasa"**,  **"image"**: **"nasa.png"**,  };  **const** templateTuit = {  ...currentUser,  **"topic"**: **"Space"**,  **"time"**: **"2h"**,  **"liked"**: **false**,  **"replies"**: 0,  **"retuits"**: 0,  **"likes"**: 0,  }  **const** tuitsSlice = *createSlice*({  **name**: **'tuits'**,  **initialState**: tuits,  **reducers**: {  createTuit(state, action) {  state.unshift({  ...action.**payload**,  ...templateTuit,  **\_id**: (**new *Date***()).getTime(),  })  }  }  });  **export const** {*createTuit*} = tuitsSlice.**actions**;  **export default** tuitsSlice.**reducer**; | | *// create an object that represents the currently*  *// logged in user which contains profile information*  *// such as username, their avatar logo, and handle.*  *// Later this will come from users login in*  *// create a template tuit object with some default*  *// values and copy over the fields userName, handle and*  *// image from the currentUser*  *// add createTuit reducer function which appends*  *// the new tuit in the payload at the beginning of the*  *// array of tuits contained in the state. Also copy*  *// all fields from templateTuit and initialize*  *// the unique identifier with a timestamp*  *// export reducer function* |

In ***whats-happening.js*** import the ***useDispatch*** hook from the redux library as shown below. Use the redux dispatcher to notify the reducer with the ***createTuit*** reducer function. Pass the ***whatsHappening*** state variable to the reducer as part of a new tuit to add to the array of tuits.

| ***whats-happening.js*** | |
| --- | --- |
| **import** {*createTuit*} **from "../tuits/tuits-reducer"**;  **import** {*useDispatch*} **from "react-redux"**;  **const** *WhatsHappening* = () => {  **let** [whatsHappening, setWhatsHappening] = *useState*(**''**);  **const** dispatch = *useDispatch*();  **const** tuitClickHandler = () => {  **const** newTuit = {  tuit: whatsHappening  }  dispatch(*createTuit*(newTuit));  }  **return** ( ... );  }  **export default** *WhatsHappening*; | *// import reducer function*  *// import dispatch hook*  *// retrieve dispatch function with hook*  *// create new tuit*  *// with text typed in textarea*  *// send tuit as action payload* |

Confirm that creating new tuits appear at the beginning of the list of tuits.

### 3.3.4 Deleting Tuits

Let's now practice deleting a tuit. To practice this, let's implement a delete tuit button as an X button on the top right corner as shown here on the right. When the user clicks on the icon, use the redux dispatcher to call a new reducer function called ***deleteTuit***, and the tuit's ID. The reducer can then remove the tuit from the tuit array, and the list of tuit would re-render minus the deleted tuit. In ***tuits-list/index.js*** implement the new delete tuit icon as shown below.

| ***tuit-item.js*** | |
| --- | --- |
| **import {*useDispatch*} from "react-redux";**  **...**  **const** dispatch = *useDispatch*();  **const** deleteTuitHandler = (id) => {  dispatch(*deleteTuit*(id));  }  **return**(  <**li className="list-group-item"**>  <**div className="row"**>  <**div className="col-auto"**>  <**img width=**{50}  **className="float-end rounded-circle"**  **src=**{**`/images/**${tuit.**image**}**`**}/>  </**div**>  <**div className="col-10"**>  <**div**>  <**i className="bi bi-x-lg float-end"**  **onClick=**{() => deleteTuitHandler(tuit.**\_id**)}></**i**>  ...  )  }  **export default** *TuitItem*; | *// import useDispatch*  *// get dispatch*  *// handle delete event*  *// pass tuit's ID to reducer*  *// X icon calls delete tuit*  *// handler when clicked* |

The dispatcher sends the action object to the tuits reducer which we handle this particular type of event in ***tuits/tuits-reducer.js*** shown below.

| ***tuits-reducer.js*** | |
| --- | --- |
| **const** tuitsSlice = *createSlice*({  **name**: **'tuits'**,  **initialState**: tuits,  **reducers**: {  deleteTuit(state, action) {  **const** index = state  .findIndex(tuit =>  tuit.**\_id** === action.**payload**);  state.splice(index, 1);  },  createTuit(state, action) {  state.unshift({  ...action.**payload**,  ...templateTuit,  **\_id**: (**new *Date***()).getTime(),  })  }  }  });  **export const** {*createTuit*, *deleteTuit*} = tuitsSlice.**actions**;  **export default** tuitsSlice.**reducer**; | *// reducer function to delete*  *// tuit looks up index of tuit*  *// from state comparing each tuit's*  *// ID with action's payload, then*  *// splices tuit from state*  *// export reducer function* |

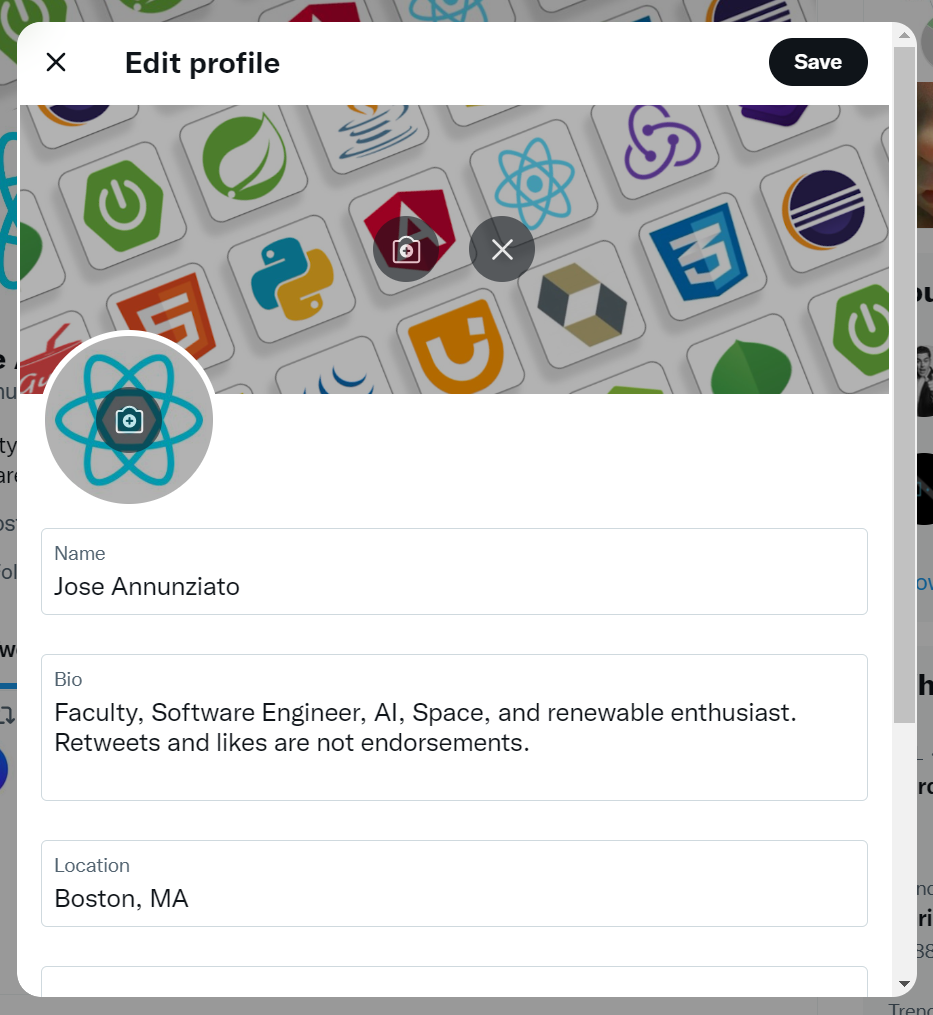


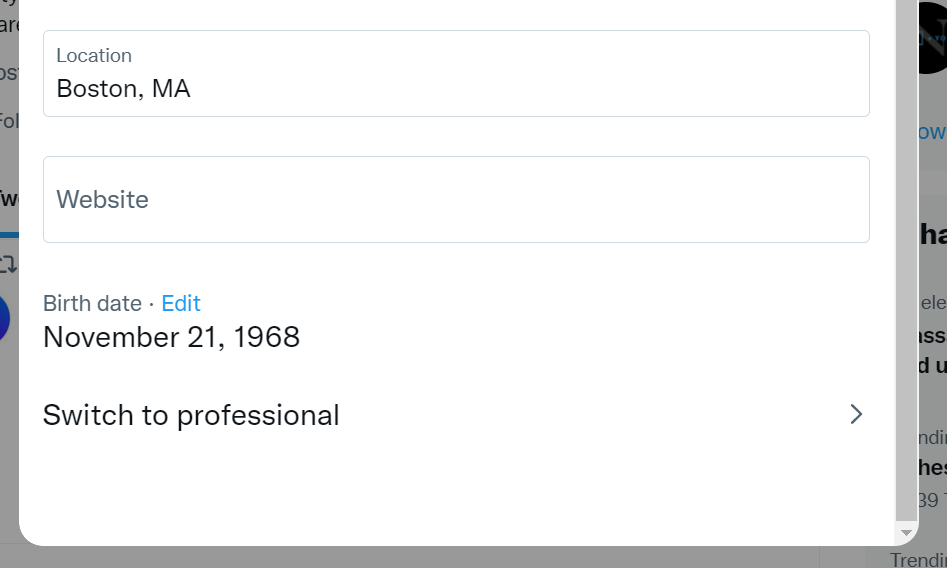
Confirm that tuits delete when you click on the new delete icon. Also confirm that you can remove new tuits as well.

# 4 Challenge (required for graduates)

As a challenge, create a ***ProfileComponent*** mapped to the ***/tuiter/profile*** route and renders as shown here on the right. The ***ProfileComponent*** should render in the center of the screen just like the ***HomeComponent*** described earlier. The ***NavigationSidebar*** should render on the left, and WhoToFollow on the right. The ***Profile*** link should highlight when the ***ProfileComponent*** is displayed. Feel free to personalize the Profile component with your own personal information or fictitious persona. Also feel free to choose your own images for the banner and avatar logo. Create a reducer called profile that keeps track of all the information about the profile such as firstName, lastName, handle, profilePicture, bannerPicture, bio, location, dateOfBirth, dateJoined, followingCount, followersCount. For instance, the profile above would be rendered from a reducer containing the following initial state.

| {  firstName: 'Jose', lastName: 'Annunziato', handle: '@jannunzi',  profilePicture: 'jose.png', bannerPicture: 'polyglot.png',  bio: 'Faculty, Software Engineer, AI, Space, and renewable enthusiast.  Retuits and likes are not endorsements.',  website: 'youtube.com/webdevtv',  location: 'Boston, MA', dateOfBirth: '7/7/1968', dateJoined: '4/2009',  followingCount: 340, followersCount: 223  } | |
| --- | --- |



Create another component called ***EditProfile*** mapped to ***/tuiter/edit-profile***, that renders as shown here on the right and shows in place of the ***Profile*** component when you click on ***Edit profile***. That is, when you click the ***Edit profile*** button in the ***Profile*** component, the path changes to ***/tuiter/edit-profile***, and the ***Profile*** component is replaced by the ***EditProfile*** component. When you click the ***Save*** button in the ***EditProfile*** component, the ***Profile*** component replaces the ***EditProfile***. The ***EditProfile*** component allows editing the user's profile, including the ***Name***, ***Bio***, ***Location***, ***Website***, and ***date of birth***. Later on it will also allow changing the ***profilePicture*** and ***bannerPicture***, but that's not required for this assignment. The ***EditProfile*** fields show the current profile information in the ***profile*** reducer. When the user clicks the ***Save*** button, the changes are saved into the ***profile*** reducer and displayed in the ***Profile*** component, and anywhere else the profile information is needed. Clicking the X button on the top left corner of the ***EditProfile*** component abandons any changes to the profile, hides the ***EditProfile*** component and displays the ***ProfileComponent*** again, without any of the changes you were making in the ***EditProfile*** component, e.g., changes are cancelled. Note: you don't have to implement image file upload, unless you want to explore that on your own. Also, ignore the horizontal line above the location input field in the wireframe

# 5 Deliverables

As a deliverable, make sure you complete the ***Labs***, ***Tuiter*** and ***Challange*** (if graduate student) sections of this assignment. All your work must be done in a branch called ***a7***. When done, add, commit and push the branch to GitHub. Deploy the new branch to Netlify and confirm it's available in a new URL based on the branch name. Submit the link to your GitHub repository and the new URL where the branch deployed to in Netlify. Here's an example on the steps:

| *Create a branch called* ***a7*** |
| --- |
| git checkout -b a7  # do all your work |

Do all your work, e.g., ***Labs*** exercises, ***Tuiter***, ***Challenge*** (graduate students)

| *Add, commit and push the new branch* |
| --- |
| git add .  git commit -am "a7 Redux fa22"  git push |

If you have ***Netlify*** configured to auto deploy, then confirm it auto deployed. If not, then deploy the branch manually.

In Canvas, submit the following

1. The new URL where your ***a7*** branch deployed to on Netlify
2. The link to your new branch in GitHub.