Managing State in React.js Application

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 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 and does/should not be shared outside those 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, profile information, e.g., ***username***, ***first name***, ***last name***, ***role***, ***logged in***, etc., might be relevant across the application, or at least relevant to a substantial portion of the application, so it might be best to be stored in application state. 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 component state. We will be using the ***Redux*** state management library to handle application state, and use ***React.js*** state and effect hooks to manage component state.

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 the previous assignment. From within IntelliJ, use ***File***, ***Open Project***, and navigate to the project directory, (***web-dev***), 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.

Installing Redux

As mentioned earlier we will be using the ***Redux*** state management library 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***, the library that integrates ***redux*** with ***React.js***. At the command line, type the following command.

```
$ npm  install  react-redux
```

Redux Examples

To learn about redux, let's create a component that will contain several simple redux examples. Create an ***index.js*** file under ***src/components/labs/redux-examples/index.js***. Copy the content below into the file.

redux-examples/index.js

```
import React from "react";
const ReduxExamples = () => {
  return(
    <div>
      <h2>Redux Examples</h2>
    </div>
  );
};
export default ReduxExamples;
```

Import the new ***ReduxExamples*** component into the ***Labs*** component so we can see how it renders as we add new examples. Reload the browser and confirm the new component renders as expected.

labs/index.js

```
import ReduxExamples from "./redux-examples";
const Labs = () => {
  return(
    <div>
      <h1>Labs</h1>
      ...
      <ReduxExamples/>
      ...
    </div>
  );
};
export default Labs;
```

Hello World

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/components/labs/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 to dynamically mutate the data as the user interacts with the application. 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/components/labs/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";                  // import useSelector hook
                                                          // from react-redux
const HelloReduxExampleComponent = () => {
 const message = useSelector((hello) => hello.message);   // extract 'Hello World' from reducer
 return(
   <h1>{message}</h1>                                     // render <h1>Hello World</h1>
 );
};

export default HelloReduxExampleComponent;
```

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           // import the component that consumes the data
 from "./hello-redux-example-component";
import hello from "./reducers/hello";       // import reducer that calculates/generates the data
import {createStore} from "redux";          // import createStore to store data from reducers
import {Provider} from "react-redux";       // import Provider which will deliver the data
const store = createStore(hello);           // create data storage

const ReduxExamples = () => {
 return(
   <Provider store={store}>                 // Provider delivers data in store to child elements,
     <div>
       <h2>Redux Examples</h2>
       <HelloReduxExampleComponent/>        // component consumes the data
     </div>
   </Provider>
 );
};
export default ReduxExamples;
```

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

Reading 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 data = [
 {
   _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 todosReducer = () => {
 return data;
}

export default todosReducer;
```

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 ***combineReducers*** to collate the various reducers into a single reducer as shown below. In ***redux-examples/index.js***, import the new ***todos*** reducers 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 the new reducer
import {Provider} from "react-redux";
import {createStore, combineReducers}           // import the combineReducers function
       from "redux";
import Todos from "./todos-component";          // import new component to render todos
const reducers =                                // combine all reducers into a single reducer
      combineReducers({hello, todos})           // each available through these namespaces

const store = createStore(reducers);            // create single store from combined reducers

const ReduxExamples = () => {
 return(
   <Provider store={store}>
```

```
      <div>
        <h2>Redux Examples</h2>
        <Todos/>                                    // render todos component (see below)
        <HelloReduxExampleComponent/>
      </div>
    </Provider>
  );
};
export default ReduxExamples;
```

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 ***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";      // import useSelector

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

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 reducers = combineReducers({hello, todos})***. The state of each reducer is now accessible through the 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>
  );
};
```

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

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. To do this we will use the ***dispatch*** hook as shown below.

todos-component.js

```
import React, {useState} from "react";
import {useDispatch, useSelector} from          // import the hook
"react-redux";

const Todos = () => {
  const todos =
        useSelector(state => state.todos);      // create a dispatch
  const [todo, setTodo] = useState({do: ''});   // handle create click event
  const dispatch = useDispatch();               // create action object
  const createTodoClickHandler = () => {        // required action type
```

```
    const action = {
      type: 'create-todo',
      todo                                    // payload
    };
    dispatch(action);                         // send action to reducers
  }
  return(
    ...
    <li className="list-group-item">
      <input
        onChange={todoChangeHandler}
        value={todo.do}
        className="form-control"/>
      <button onClick={createTodoClickHandler}  // create button notifies event handler
        className="btn btn-primary">            // when clicked to create a new todo
        Create New Todo
      </button>
    </li>
    ...
  );
};
export default Todos;
```

All the reducers will be invoked with the same action object. We'll handle the ***create-todo*** event in the todos reducer as shown below. We'll append the new todo passed through in the action to the end of the new state. Refresh the website and confirm you can type and create a new todo that appears at the top of all todos.

todos-reducer.js

```
const todosReducer = (state = data,
action) => {
 switch (action.type) {
   case 'create-todo':                  // handle create-todo action type
     return [                           // create new state as array
       ...state,                        // containing old todos
       action.todo                      // and appending the new todo at the end of the array
     ]                                  // return new state
   default:
     return state;
 }
}
```

Deleting from application state

We can delete todos by filtering out the deleted todo from the current array of todos. To start let's add a delete to all the todos and bind a click event handled by the event handler shown below.

todos-component.js

```
const Todos = () => {
  ...
  const dispatch = useDispatch();
  const deleteTodoClickHandler = (todo) => {   // delete todo event handler accepts todo
    const action = {                           // create new action
```

```
      type: 'delete-todo',                    // with type 'delete-todo'
      todo                                    // include todo to remove
    };
    dispatch(action);                         // send action to all reducers
  }
  ...
  return(
    ...
    <li className="list-group-item">
      {todo.do}
      <button onClick={() =>                  // if user clicks on Delete button
        deleteTodoClickHandler(todo)}         // call event handler passing todo instance
        className="btn btn-danger float-end">
        Delete
      </button>
    </li>
                          ...
 );
};
export default Todos;
```

The dispatch will send the action object to all reducers, but only the todos reducer is going to pay any attention to the action type ***delete-todo***. The reducer will create a brand new state that contains all the todos minus the one being deleted. This is achieved using a filter that skips over the deleted todo. Refresh the Website and confirm that you can delete todos.

todos-reducer.js

```
const todosReducer = (state = data, action) => {
  switch (action.type) {
    case 'delete-todo':                       // todos reducer handles 'delete-todo' event
      return state                            // new state is filtered todo out from todos
             .filter(todo =>
               todo !== action.todo);
    case 'create-todo':
    ...
    default:
      return state;
  }
}
```

Updating application state

Let's practice changing something in a reducer. To do this, let's add a ***done*** flag we can change easily with a checkbox. Add a ***done*** property to the todo initialization as shown below

todos-component.js

```
const Todos = () => {
  const [todo, setTodo] =
        useState({do: '', done: false});      // add done property
  const dispatch = useDispatch();
  const updateTodoClickHandler = (todo) => {  // handle checkbox click
    const action = {                          // create new action
```

```
      type: 'update-todo',                        // with update-todo event
      todo                                        // include todo object
    };
    dispatch(action);                             // send to all reducers
  }
  return(
    ...
    {
      todos.map(todo =>
        <li className="list-group-item">
          <input checked={todo.done}              // create a checkbox, show checked if done=true
            onChange={(event) =>                  // if checkbox changes, update done with
              updateTodoClickHandler(             // checkbox's value
                {...todo,                         // copy old state
                 done: event.target.checked})}    // overwrite done with target.checked
            type="checkbox"/>
          {todo.do}
    ...
        </li>)
    }
    ...
 );
};
```

Handle the ***update-todo*** event by rebuilding the array from the old and new versions. As you iterate over the todos, keep the old ones if they are not the one being updated, and keep the new one if it's the one passed in ***action.todo***. Refresh the screen and confirm you can update the ***done*** checkbox.

todos-reducer.js

```
const todosReducer = (state = data, action) => {
 switch (action.type) {
   case 'update-todo':
     const newTodos = state.map(todo => {
       const newTodo = todo._id === action.todo._id ? action.todo : todo;
       return newTodo;
     });
     return newTodos;
   case 'delete-todo':
     return state.filter(todo => todo !== action.todo);
   case 'create-todo':
     const newTodo = {
       ...action.todo,
       _id: (new Date()).getTime() + ""
     };
     return [
       ...state,
       newTodo
     ]
   default:
     return state;
 }
}

export default todosReducer;
```

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/components/tuiter***.

Refactor Routes

Refactor routing to use the latest version of React Router which can define nested routes in a central place instead of in multiple files. Use the code snippet below as a guide. The routes below define a top level route that contain nested routes labs, hello world, and tuiter. Within the tuiter route there are several routes related to the tuiter application.

App.js

```
function App() {
 return (
   <BrowserRouter>
     <div className="container">
       <Routes>
         <Route path="/">                              // root route contains top level routes
           <Route path="labs"                          // labs route loads
                  element={<Labs/>}/>                  // Labs screen
           <Route path="hello"                         // hello route loads
                  element={<HelloWorld/>}/>            // HelloWorld screen
           <Route path="tuiter"                        // tuiter route loads
                  element={<Tuiter/>}>                 // Tuiter screen
             <Route index                              // index is default route which loads
                    element={<HomeScreen/>}/>          // default screen HomeScreen
             <Route path="explore"                     // explore route loads
                    element={<ExploreScreen/>}/>       // ExploreScreen screen
             <Route path="notifications"               // notifications route loads
                    element={<NotificationScreen/>}/>  // NotificationScreen
             ...                                       // declare other routes to load
           </Route>                                    // other screens
         </Route>
       </Routes>
     </div>
   </BrowserRouter>
 );
}
```

The ***Tuiter*** screen can be refactored as shown below in ***tuiter/index.js*** consisting of a screen split in three columns. The left column renders the ***NavigationSidebar*** component. The center column defines an ***Outlet*** which will render any of the components defined under the ***Tuiter*** route defined in ***App.js***. If the path is ***/tuiter*** then ***Outlet*** renders the default screen ***Home***, if the path is ***/tuiter/explore*** then ***Outlet*** renders ***ExploreScreen***, if the path is ***/tuiter/notifications*** then ***Outlet*** renders ***NotificationsScreen***. The right column will render the ***WhoToFollowList*** component implemented in the next section.

tuiter/index.js

```
import {Outlet} from "react-router-dom";
import NavigationSidebar from "./navigation-sidebar";
```

```
import "./tuiter.css";

const Tuiter = () => {
  return (
    <div className="row mt-2">
      <div className="col-2 col-lg-1 col-xl-2">
        <NavigationSidebar/>
      </div>
      <div className="col-10 col-lg-7 col-xl-6">
        <Outlet/>
      </div>
      <div className="d-none d-lg-block col-lg-4 col-xl-4">
        <h2>Who to follow</h2>
      </div>
    </div>
  );
};
export default Tuiter;
```

Rendering WhoToFollowList

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 whoJson from '../data/who.json';     // import data from JSON file

const whoReducer = (state = whoJson) => {   // initialize the reducer's state
  return(state);                            // return the single static state
};

export default whoReducer;
```

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 whoReducer from "./reducers/who-reducer";  // import the reducer
import {createStore} from "redux";                // import createStore
import {Provider} from "react-redux";             // import the Provider
const store = createStore(whoReducer);            // create the store from the reducer
const Tuiter = () => {
  return (
    <Provider store={store}>                      // provide the store to all child components
    ...
    </Provider>
```

```
  );
};
export default Tuiter;
```

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 state from the store.

who-to-follow-list/index.js

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

Import the ***WhoToFollowList*** component in the ***Tuiter*** component so it renders in the right most column as shown below. Refresh the tuiter path and confirm it renders as expected.

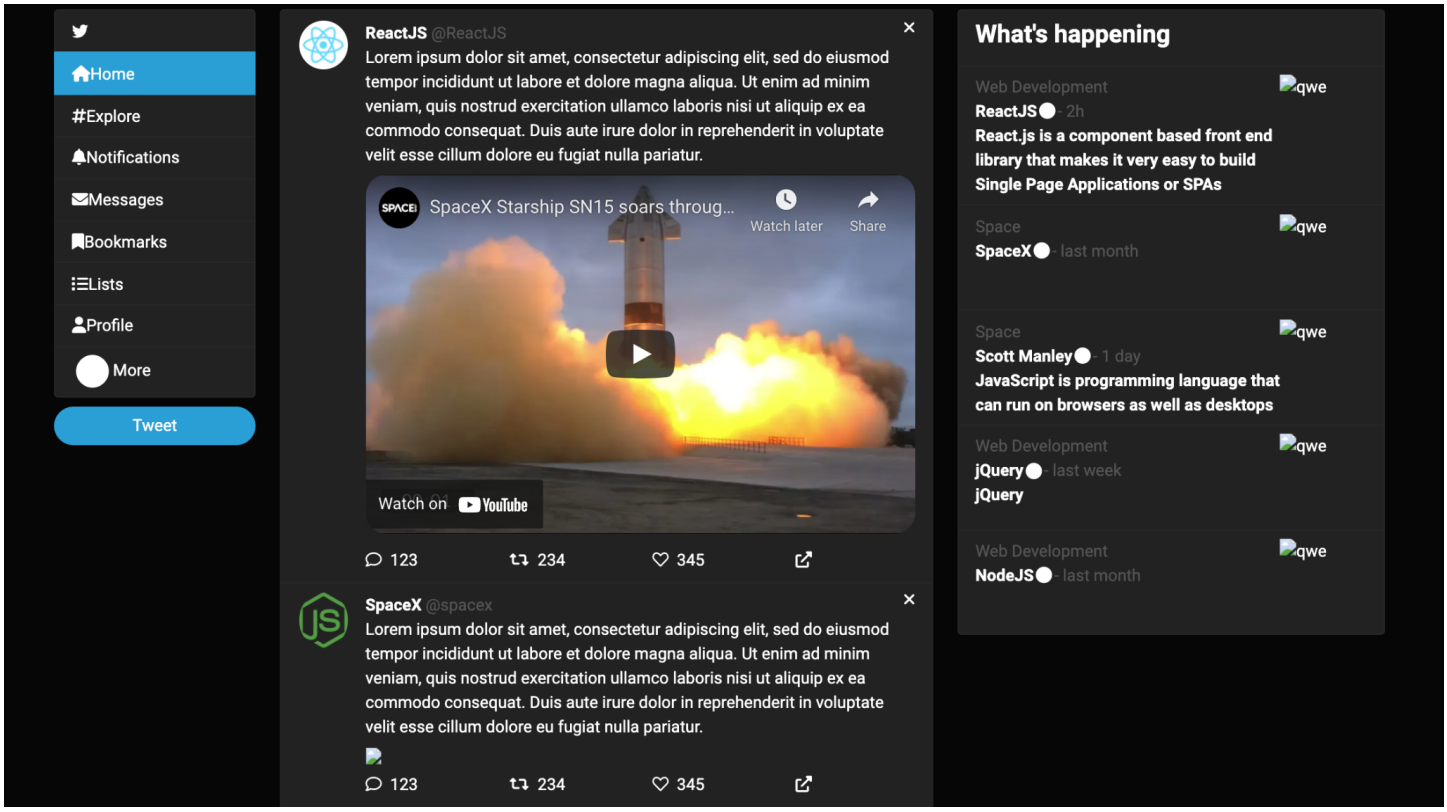
tuiter/index.js

```
...
import WhoToFollowList                  // import who to follow list
       from "./who-to-follow-list";
...
const Tuiter = () => {
  return (
    <Provider store={store}>
      <div className="row mt-2">
        ...
        <div className="...">
          <h2>Who to follow</h2>        // replace placeholder with actual
          <WhoToFollowList/>            // who to follow list components
        </div>
      </div>
    </Provider>
  );
};
```

Implementing the Home Screen

Let's create a new ***Home*** screen as shown below in the wireframe. If you have already implemented the Home screen, feel free to reuse your implementation. You might need to refactor it somewhat for the purposes of this assignment. As configured in the router earlier the home screen will appear in the center column with the navigation bar on the left and the ***WhoToFollowList*** component on the right as implemented in the previous section. The main content at the center will be a new component called ***TuitList*** we'll implement here. If you've

already implemented ***TuitList***, then this section will walk you how to refactor it to use a reducer as a data source. In my source repository I provide sample code for several of the components described here. Read on.



Create a new ***HomeScreen*** component in ***src/components/tuiter/home-screen/index.js*** as shown below with a placeholder for the list of tuits we'll soon render.

home-screen/index.js

```
const HomeScreen = () => {
  return(
    <div>
      <h2>Home</h2>
      <h3>Tuit list</h3>              // implemented below
    </div>
  )
}
export default HomeScreen;
```

rate file called ***src/components/tuiter/tuit-list/index.js***. Import an array of tuits from ***tuits.json***, iterate over the array, and render each tuit as a ***TuitListItem*** as shown below. [You can get a sample copy of tuits.json from my code repository](#).

tuit-list.js

```
import React from "react";
import tuits from "./tuits.json";          // import the tuits
import TuitListItem                        // import TuitListItem
       from "./tuit-list-item";
```

```
import './tuits.css';

const TuitList = () => {
  return (
    <ul className="ttr-tuits list-group">
      {
        tuits.map && tuits.map(tuit =>      // iterate over each tuit and
          <TuitListItem key={tuit._id}      // render it as a TuitListItem
                        tuit={tuit}/>)
      }
    </ul>
  );
}

export default TuitList;
```

Implement ***TuitListItem*** above in a file called ***src/components/tuiter/tuit-list/tuit-list-item.js*** so that it renders a tuit object as shown in the wireframe above. Feel free to reuse HTML snippets from earlier assignments to render and style individual tuits. Once you're done implementing the ***TuitList*** and ***TuitListItem*** components, import them as necessary into the ***HomeScreen*** component replacing the `<h3>Tuit list</h3>` with the actual ***TuitList*** component. Verify that the tuits render when you navigate to the home screen.

Rendering a list of Tuits with Redux

The earlier ***TuitList*** implementation imported ***tuits.json*** and iterated over 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 tuits. Let's move the ***tuits.json*** file into ***src/components/tuiter/data/tuits.json***, and then implement the reducer shown below in ***reducers/tuits-reducer.js***.

tuits-reducer.js

```
import tuits from "../data/tuits.json";

const tuitsReducer = (state = tuits) => {
  return tuits;
}

export default tuitsReducer;
```

Combine the ***tuitsReducer*** with the ***whoReducer*** you already used in a previous exercise. Use the code below as a guide to combine reducers in ***tuiter/index.js***.

tuiter/index.js

```
import whoReducer from "./reducers/who-reducer";
import tuitsReducer from "./reducers/tuits-reducer";   // load tuits reducer
import {combineReducers, createStore} from "redux";    // load combine reducers
import {Provider} from "react-redux";
const reducer = combineReducers({                      // combine reducers into single reducer
  tuits: tuitsReducer, who: whoReducer                 // namespace data from each reducer
```

```
});
const store = createStore(reducer);                    // create single store from all reducers
const Tuiter = () => {
  return (
    <Provider store={store}>
    ...
    </Provider>
  );
};
export default Tuiter;
```

Now we need to refactor ***tuit-list.js*** so that it retrieves the tuits from the store instead of the local file as shown below. Refresh the screen and confirm that the tuits display as expected.

tuit-list/index.js

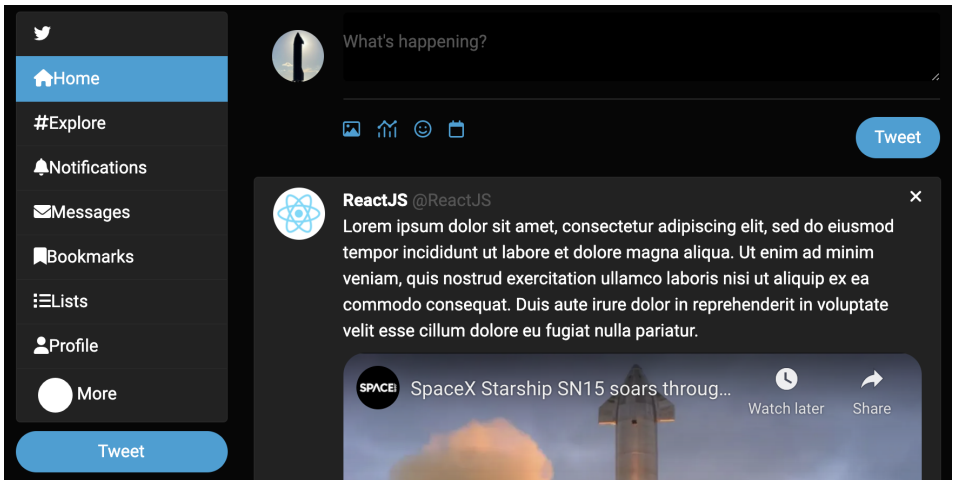
```
import TuitListItem from "./tuit-list-item";
import tuits from "./tuits.json";                       // replace getting tuits from a file
import {useSelector} from "react-redux";                // to getting tuits from the store
import './tuits.css';

const TuitList = () => {
  const tuits = useSelector(                            // get the tuits from the state in the store
    state => state.tuits);
```

We also need to refactor ***who-to-follow-list.js*** so that it retrieves the ***who*** array from the new ***who*** state namespace

who-to-follow-list.js

```
const WhoToFollowList = () => {
 const who = useSelector(state => state.who);
```

Working with Forms



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. Implement the ***textarea***, icons and ***Tuit*** button as shown here, in a new file called ***whats-happening.js*** using 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 notify 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.

whats-happening.js

```
import React, {useState} from "react";            // get the useState hook

const WhatsHappening = () => {
  let [whatsHappening, setWhatsHappening]         // create whatsHappening state variable
      = useState('');
  const tuitClickHandler = () => {                // handle tuit button click
    console.log(whatsHappening);                  // just print whatsHappening
  }                                               // state variable for now
  return (
    <>
      <textarea value={whatsHappening}            // show current whatsHappening in textarea
                onChange={(event) =>              // if it changes, update whatsHappening
        setWhatsHappening(event.target.value)}>   // with textarea's value
      </textarea>
      <button onClick={tuitClickHandler}>         // notify Tuit button click
        Tuit
      </button>
    </>
  );
}
export default WhatsHappening;
```

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

home-screen/index.js

```
import WhatsHappening                 // import WhatsHappening
       from "./whats-happening";
const HomeScreen = () => {
 return(
   <div>
     <WhatsHappening/>                // include WhatsHappening in the HomeScreen
     <TuitList/>
   </div>
 )
}
```

Creating Tuits

Let's practice creating new tuits. In ***whats-happening.js*** import the ***useDispatch*** hook from the redux library as shown below. Use the redux dispatcher to notify reducers of a ***create-tuit*** event. 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 {useDispatch}                        // import the useDispatch redux hook
```

```
      from "react-redux";
const WhatsHappening = () => {
 let [whatsHappening, setWhatsHappening]
     = useState('');
 const dispatch = useDispatch();          // use the hook to get a dispatcher
 const tuitClickHandler = () => {         // use the dispatcher to notify reducer of new
   dispatch({type: 'create-tuit',         // tuit including text written in textarea
     tuit: whatsHappening                 // saved in whatsHappening state variable
   });
 }
 return (...);
}
export default WhatsHappening;
```

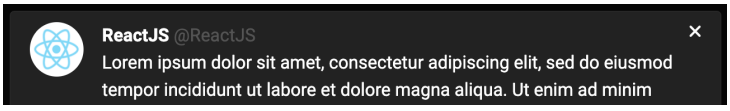
In the reducer implemented in ***reducers/tuits.js***, handle the ***create-tuit*** event by creating a brand new tuit object as shown below. Add default attributes such as ***_id***, ***topic***, ***userName***, etc as shown below. Copy the tuit in the action object that includes the ***whatsHappening*** attribute sent by the dispatcher shown above.

tuits-reducer.js

```
const tuitsReducer =
  (state = tuits, action) => {
  switch (action.type) {                  // add a switch to handle different event types
    case 'create-tuit':                   // handle first event type create-tuit
      const newTuit = {                   // create new tuit object with several default
        tuit: action.tuit,                // values for _id, postedBy, userName, etc.
        _id: (new Date()).getTime() + '',
        postedBy: {
          "username": "ReactJS"
        },
        stats: {
          retuits: 111,
          likes: 222,
          replies: 333
        }
      }
      return [
        newTuit,                          // append new tuit object at beginning of
        ...state,                         // array of tuit. Put older tuits at the end
      ];                                  // of the tuits array
    default:                              // it's a good practice to always have default
      return tuits                        // case returning the current state
  }
}
```

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

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. Add a fontawesome icon to the left of the tuit as shown in the code below. When the user clicks on the icon, use the

redux dispatcher to notify the reducer with a ***delete-tuit*** event, and the tuit itself. The reducer can then remove the tuit from the tuit array, and the list of tuit would re-render minus the deleted tuit. In ***tuit-list/index.js*** implement the new delete tuit icon as shown below.

tuit-list-item/index.jx

```
import {useDispatch} from "react-redux";     // import useDispatch
const TuitListItem = ({tuit}) => {
  const dispatch = useDispatch();            // get dispatcher to send message to reducer
  const deleteTuit = (tuit) => {             // handle delete tuit click event
    dispatch({type: 'delete-tuit', tuit})    // notify redux reducer with delete-tuit event and
  };                                         // deleted tuit
  return(
    ...
    <i onClick={() =>                        // create new remove icon on top, right corner of
       deleteTuit(tuit)}                     // each tuit. Bind click event with click handler
       className="fas fa-remove fa-2x
                  fa-pull-right"></i>
    ...
  );
}
export default TuitListItem;
```

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

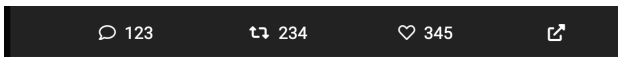
tuits-reducer.js

```
const tuitsReducer =
  (state = tuits, action) => {
  switch (action.type) {
    case 'delete-tuit':                          // handle new action type to delete a tuit
      return state.filter(                       // calculate new state by
        tuit => tuit._id !== action.tuit._id);   // filtering out the tuit we deleted in the
    ...                                          // user interface
    default:
      return tuits
  }
}
```

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

Modifying Tuits

Now let's take a look at modifying a tuit. To practice modifying a tuit, let's update its numbers of likes when someone clicks the heart link below the tuit. If someone clicks the heart icon, the number of likes should increase by one and the heart should turn red. If the user clicks the heart again, then it's interpreted as unliking the tuit, the count goes back down by one and the heart is no longer red as shown below. Implement these statistics in a separate ***TuitStats*** component implemented in ***tuit-stats.js***



To implement this we can keep track on whether the tuit is liked or not in a state variable ***liked*** part of the current ***tuit***. If ***liked*** is true, then we'll render a red heart, otherwise we'll render a normal heart. If the heart is clicked we'll update the state of the likes count in the stats property. Use the code below as a guide to update the likes and liked property in ***tuit-stats.js***.

tuit-stats.js

```
import {useDispatch} from "react-redux";

const TuitStats = ({tuit}) => {
  const dispatch = useDispatch();              // get redux dispatch to notify state change
  const likeTuit = () => {                     // handle like click event
    dispatch({type: 'like-tuit', tuit});       // notify reducer tuit was liked. send action
  };                                           // object with type and tuit
  return (
    ...
    <span onClick={likeTuit}>                  // if use clicks the heart icon
    {                                          // notify click handler
      tuit.liked &&                            // if tuit is liked, render red heart
      <i className="fas fa-heart me-1"
         style={{color: 'red'}}></i>
    }
    {
      !tuit.liked &&                           // otherwise render regular heart
      <i className="far fa-heart me-1"></i>
    }
    {tuit.stats && tuit.stats.likes}           // also render number of likes
    </span>
    ...
  );
}
export default TuitStats;
```

The dispatched action is sent to all reducers which is handled by the tuit reducer below. It the action.type is 'like-tuit' a new state is for the tuit's liked is calculated. We iterate over all the tuits looking for the liked tuit. If the tuit was already liked, then we unlike it and decrease liked counter. If the tuit had not been liked yet, the we set it's liked flag to true, and increment its liked counter. In ***reducers/tuits.js***, add the 'liked-tuit' case below.

tuits-reducer.js

```
const tuitsReducer =
  (state = tuits, action) => {
  switch (action.type) {
    case 'like-tuit':                          // handle action.type 'like-tuit' event
      return state.map(tuit => {               // calculate a new state
        if(tuit._id === action.tuit._id) {     // if it's the tuit we liked
          if(tuit.liked === true) {            // and it's already liked
            tuit.liked = false;                // then unlike it
            tuit.stats.likes--;                // and reduce likes count
          } else {                             // otherwise
            tuit.liked = true;                 // like the tuit
            tuit.stats.likes++;                // and increment its like count
          }
          return tuit;                         // include new tuit changes in array of tuits
        } else {                               // otherwise
          return tuit;                         // keep the old tuit object
```

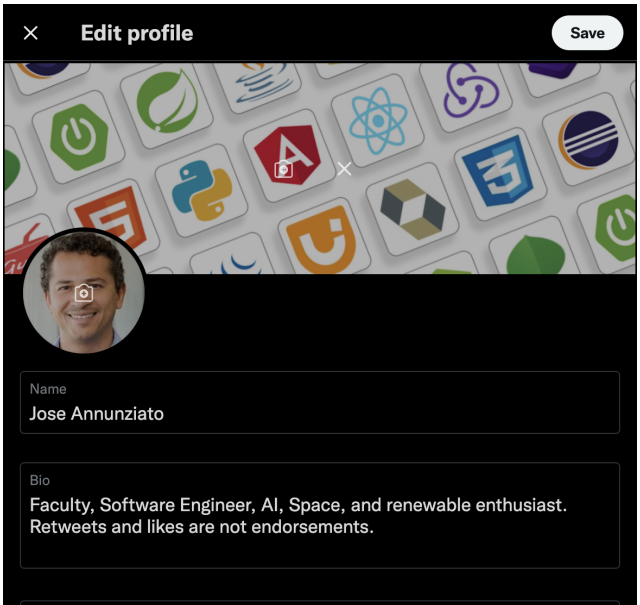
```
        }
    });
```

Refresh the application and confirm that clicking on a like heart icon toggles it to a red heart and clicking it again toggles it back to a normal heart icon. Also confirm that the likes counter updates correctly up and down as you click the heart icon on and off.

Challenge (required for graduates)

As a challenge, create a ***ProfileScreen*** mapped to the ***/tuiter/profile*** route. The ***ProfileScreen*** should render just like the ***HomeScreen*** described earlier, in the top center a new ***Profile*** component renders as shown here on the right. The ***NavigationSidebar*** should render on the left, just like in the HomeScreen, but the ***Profile*** link should highlight when the ***ProfileScreen*** 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. 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.

```
const profileData: {
  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: 312,    followersCount: 180
}
```



Create another component called ***EditProfile*** that renders as shown here on the right. When you click the ***Edit*** profile button in the ***Profile*** component, it 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***. 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 Profile 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.

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 sp22"
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.