Exercise 6: Managing State in React.js Application

1 Introduction

In this exercise, 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 data structures 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 to handle application state, and use **React. js** state and effect hooks to manage component state.

2 Exercises

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 exercise. After you work through the examples you will apply the skills while creating a *Tuiter* on your own. Do all your work in a new directory called *e6*.

2.1 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* 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 Exercise 6 component

To get started, create an **exercise6** component that will host all the works in this exercise. Then import the component into the **exercises** component created earlier. While you're in the **exercises** component, add routes so that each exercise will appear in its own screen when you navigate to exercises and then to **/e6**. That is, make

the **exercise5** component the default element that renders when navigating to the **exercises** path and map **exercise6** to the **/e6** path. You'll need to change the **exercises** 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=**{**Exercises**/**>**}/>. Use the code snippets below as a guide.

```
src/ exercises
                               src/ exercises / index. js
                                                                              src/nav.js
/e6/index.js
import React from "react";
                               import Nav from "../nav";
                                                                              import {Link} from "react-router-dom";
                               import Exercise5 from "./e5";
                               import Exercise6 from "./e6";
const Exercise6 = () => {
                                                                              function Nav() {
                               import {Routes, Route} from "react-router";
                                                                               return (
 return(
                                                                                 <div>
                                                                                   <Link to="/"> Exercises </Link>
    <h1>Exercise 6</h1>
                               function Exercises() {
   </>
                                return (
                                                                                   <Link to="/"> Exercise 5</Link>
                                                                                  <Link to="/e6"> Exercise 6</Link>
);
                                  <div>
                                                                                   <Link to="/tuiter">Tuiter</Link>
                                   <Nav/>
export default Exercise6;
                                   <Routes>
                                                                                 </div>
                                     <Route index
                                                                               );
                                      element={< Exercise5 />}/>
                                     <Route path="e6"
                                      element={< Exercise6 />}/>
                                                                              export default Nav;
                                    </Routes>
                                  </div>
                                );
                               export default Exercises;
```

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 *e6/redux-examples/index.js* as shown below. Import the new redux examples component into the *exercise6* 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
                                                    e6/index.js
                                                    import React from "react";
import React from "react";
                                                    import ReduxExamples from "./redux-examples";
const ReduxExamples = () => {
                                                    const Exercise6 = () => {
                                                     return(
      <h2>Redux Examples</h2>
    </div>
                                                        <h1>Exercise 6</h1>
 );
                                                         <ReduxExamples/>
                                                     );
export default ReduxExamples;
                                                    };
                                                    export default Exercise6;
```

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 e6/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.

```
e6/redux-examples/reducers/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 e6/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.

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 which will deliver the data
import {Provider} from "react-redux";
const store = createStore(hello);
                                           // create data storage
const ReduxExamples = () => {
 return(
   <Provider store={store}>
                                           // Provider delivers data in store to child elements,
       <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.

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.

```
Reducers/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.

```
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 the new reducer
```

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

```
redux-examples/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>
    todos.map(todo =>
                                        // iterate over todos array and render a
          // line item element for each todo object
                                        // display do property containing the todo text
           {todo.do}
         }
    </>
);
};
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 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.

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

Create a future of abundance

Accelerate the world's transition to sustainable energy

Reduce space transportation costs to become a spacefaring civilization

```
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(
                                           // add a new line item at the top
  // containing an input field to type todo
        className="form-control"
        value={todo.do}
        onChange={todoChangeHandler}/>
                                           // handle keystrokes to update component state
                                           // update field with latest state value
    . . .
);
};
export default Todos;
```

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

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 Create a future of abundance Accelerate the world's transition to sustainable energy Reduce space transportation costs to become a spacefaring civilization Create a future of abundance

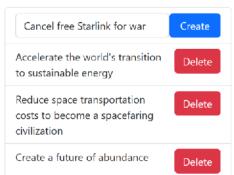
```
todos-component.js
import React, {useState} from "react";
import {useDispatch, useSelector}
                                                 // useDispatch hook to call reducers
 from "react-redux";
                                                 // import reducer function exported by
import {addTodo}
 from "./reducers/todos-reducer";
                                                 // todos-reducer
const Todos = () => {
const todos = useSelector(state => state.todos);
const [todo, setTodo] = useState({do: ''});
                                                 // get distacher to invoke reducer functions
const dispatch = useDispatch();
const createTodoClickHandler = () => {
                                                 // handle click event of button
                                                 // call reducer function passing new todo
  dispatch(addTodo(todo))
                                                 // as the payload in the action object
return(
  <h3>Todos</h3>
  // new button to add new todo
      <button onClick={createTodoClickHandler}</pre>
              className="btn btn-primary w-25
                                                 // calls function to handle click event
```

```
float-end">
Create</button>
<input onChange={todoChangeHandler}
value={todo.do}
className="form-control w-75"/>
...
```

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.

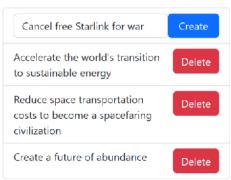
Todos



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

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

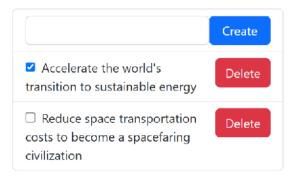


```
todos-reducer.js
reducers: {
addTodo(state, action) {
  state.push({
     _id: (new Date()).getTime(),
    do: action.payload.do,
    done: false
  });
},
deleteTodo(state, action) {
                                                          // new deleteTodo function extracts
   const index = action.payload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

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



```
return(
  <>
   <h3>Todos</h3>
    todos.map((todo, ndx) =>
     key={todo._id}
          className="list-group-item">
       <button onClick={() =>
        deleteTodoClickHandler(ndx)}
             className="btn btn-danger
                       float-end ms-2">
       Delete
     </button>
     <input type="checkbox"</pre>
                                              // new checkbox which is checked
            checked={todo.done}
                                              // if todo.done is true
            onChange={() =>
                                              // if user changes checkbox, we'll pass the
                                              // todo to reducer function to update todo's
            toggleTodoDone(todo)}
            className="me-2"/>
                                              // state
     {todo.do}
   )
}
```

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.