Managing State in React.js Application

1 Introduction

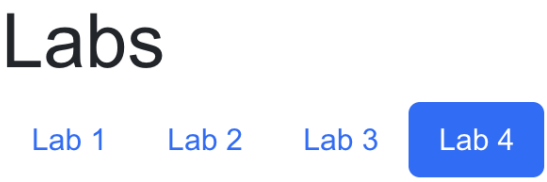
In this assignment we are going to practice working with application and component level state. ***State*** is the collection of data values stored in the various constants, variables and data structures in an application. ***Application state*** is data that is relevant across the entire application 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 user could be stored in a profile, e.g., ***username***, ***first name***, ***last name***, ***role***, ***logged in***, etc., and it might be relevant across the entire 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 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 ***Kanbas*** on your own. Using ***IntelliJ***, ***VS Code***, or your favorite IDE, open the project you created in previous assignments. ***Include all the work in the Labs section as part of your final deliverable***. Do all your work in a new branch called ***a4*** 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 Create an Lab4 Screen

To get started, create an ***Lab4*** scree that will host all the exercises in this assignment. Import the component into the ***Labs*** screen created in an earlier assignment and add a route to navigate to ***Lab4***. Style the links as Bootstrap pills.



2.2 Handling User Events

2.2.1 Handling Click Events

The ***onClick*** attribute can be used to declare a function that handles clicks. The example below calls function ***hello*** when you click the ***Click Hello*** button. Add the component to ***Lab4*** and confirm it behaves as expected.

src/Labs/Lab4/ClickEvent.tsx

```
const hello = () => {
  alert("Hello World!");
};
const lifeIs = (good: string) => {
  alert(`Life is ${good}`);
};
```

```
// declare a function to handle the event
```

```tsx
export default function ClickEvent() {
  return (
    <div id="wd-click-event">
      <h2>Click Event</h2>
      <button onClick={hello} id="wd-hello-world-click">
        Hello World!</button>
      <button onClick={() => lifeIs("Good!")}
              id="wd-life-is-good-click">
        Life is Good!</button>
      <button onClick={() => {
                hello();
                lifeIs("Great!");
              }} id="wd-life-is-great-click">
        Life is Great!
      </button>
      <hr/>
    </div>
);}
```

```tsx
// configure the function call

// wrap in function if you need to pass parameters

// wrap in {} if you need more than one line of code
// calling hello()
// calling lifeIs()
```

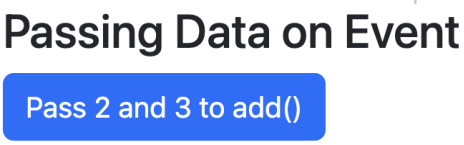
2.2.2 Passing Data when Handling Events

When handing an event, sometimes we need to pass parameters to the function handling the event. Make sure to wrap the function call in a ***closure*** as shown below. The example below calls ***add(2, 3)*** when the button is clicked, passing arguments ***a*** and ***b*** as ***2*** and ***3***. If you do not wrap the function call inside a closure, you risk creating an infinite loop. Add the component to ***Lab4*** and confirm it works as expected.

src/Labs/Lab4/PassingDataOnEvent.tsx

```tsx
const add = (a: number, b: number) => {
  alert(`${a} + ${b} = ${a + b}`);
};
export default function PassingDataOnEvent() {
  return (
    <div id="wd-passing-data-on-event">
      <h2>Passing Data on Event</h2>
      <button onClick={() => add(2, 3)}
              // onClick={add(2, 3)}
              className="btn btn-primary"
              id="wd-pass-data-click">
        Pass 2 and 3 to add()
      </button>
      <hr/>
    </div>
);}
```

```tsx
// function expects a and b

// use this syntax
// and not this syntax.
// Otherwise you risk
// creating an infinite loop
```



2.2.3 Passing Functions as Parameters

In JavaScript, functions can be treated as any other constant or variable, including passing them as parameters to other functions. The example below passes function ***sayHello*** to component ***PassingFunctions***. When the button is clicked, ***sayHello*** is invoked.

src/Labs/Lab4/PassingFunctions.tsx

```tsx
export default function PassingFunctions(
 { theFunction }: { theFunction: () => void }) {
  return (
    <div>
      <h2>Passing Functions</h2>
      <button onClick={theFunction} className="btn btn-primary">
        Invoke the Function
      </button>
      <hr/>
    </div>
);}
```

```tsx
// function passed in as a parameter


// invoking function with no arguments
```

Include the component in ***Lab4***, declare a ***sayHello*** callback function, pass it to the ***PassingFunctions*** component, and confirm it works as expected.

src/Labs/Lab4/index.tsx

```
import PassingFunctions from "./PassingFunctions";
export default function Lab4() {
  function sayHello() {
    alert("Hello");
  }
  return (
    <div id="wd-passing-functions">
      <h2>Lab 4</h2>
      ...
      <PassingFunctions theFunction={sayHello} />
    </div>
);}
```

```
// import the component

// implement callback function




// pass callback function as a parameter
```

2.2.4 The Event Object

When an event occurs, JavaScript collects several pieces of information about when the event occurred, formats it in an ***event object*** and passes the object to the event handler function. The ***event object*** contains information such as a timestamp of when the event occurred, where the mouse was on the screen, and the DOM element responsible for generating the event. The example below declares event handler function ***handleClick*** that accepts an ***event object e*** parameter, removes the ***view*** property and replaces the ***target*** property to avoid circular references, and then stores the event object in variable ***event***. The component then renders the JSON representation of the event on the screen. Include the component in ***Lab4***, click the button and confirm the event object is rendered on the screen.

src/Labs/Lab4/EventObject.tsx

```
import React, { useState } from "react";
export default function EventObject() {
  const [event, setEvent] = useState(null);
  const handleClick = (e: any) => {
    e.target = e.target.outerHTML;
    delete e.view;
    setEvent(e);
  };
  return (
    <div>
      <h2>Event Object</h2>
      <button onClick={(e) => handleClick(e)}
        className="btn btn-primary"
        id="wd-display-event-obj-click">
        Display Event Object
      </button>
      <pre>{JSON.stringify(event, null, 2)}</pre>
      <hr/>
    </div>
);}
```

```
// import useState
// (more on this later)
// initialize event
// on click receive event
// replace target with HTML
// to avoid circular reference
// set event object
// so it can be displayed



// button that triggers event
// when clicked passes event
// to handler to update
// variable

// convert event object into
// string to display
```

Event Object

Display Event Object


```
{
  "_reactName": "onClick",
  "_targetInst": null,
  "type": "click",
  "nativeEvent": {
    "isTrusted": true
  },
  "target": "<button id=\"event-button\"
  "currentTarget": null,
  "eventPhase": 3,
  "bubbles": true,
  "cancelable": true,
  "timeStamp": 1576.8999999761581,
  "defaultPrevented": false,
  "isTrusted": true,
  "detail": 1,
  "screenX": 226,
  "screenY": 244,
```


2.3 Managing Component State

Web applications implemented with React.js can be considered as a set of functions that transform a set of data structures into an equivalent user interface. The collection of data structures and values are often referred to as an application ***state***. So far we have explored React.js applications that transform a static data set, or state, into a static user interface. We will now consider how the state can change over time as users interact with the user interface and how these state changes can be represented in a user interface.

Users interact with an application by clicking, dragging, and typing with their mouse and keyboard, filling out forms, clicking buttons, and scrolling through data. As users interact with an application they create a stream of events that can

be handled by a set of event handling functions, often referred to as ***controllers***. Controllers handle user events and convert them into changes in the application's state. Applications render application state changes into corresponding changes in the user interface to give users feedback of their interactions. In Web applications, user interface changes consist of changes to the DOM.

2.3.1 Use State Hook

Updating the DOM with JavaScript is slow and can degrade the performance of Web applications. React.js optimizes the process by creating a ***virtual DOM***, a more compact and efficient version of the real DOM. When React.js renders something on the screen, it first updates the virtual DOM, and then converts these changes into updates to the actual DOM. To avoid unnecessary and slow updates to the DOM, React.js only updates the real DOM if there have been changes to the virtual DOM. We can participate in this process of state change and DOM updates by using the ***useState*** hook. The ***useState*** hook is used to declare ***state*** variables that we want to affect the DOM rendering. The syntax of the ***useState*** hook is shown below.

```
const [stateVariable, setStateVariable] = useState(initialStateValue);
```

The ***useState*** hook takes as argument the initial value of a ***state variable*** and returns an array whose first item consists of the initialized state variable, and the second item is a ***mutator*** function that allows updating the state variable. The array destructor syntax is commonly used to bind these items to local constants as shown above. The mutator function not only changes the value of the state variable, but it also notifies React.js that it should check if the state has caused changes to the virtual DOM and therefore make changes to the actual DOM. The following exercises introduce various use cases of the ***useState***.

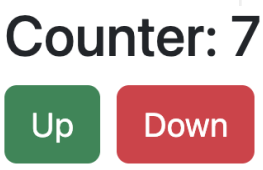
2.3.2 Integer State Variables

To illustrate the point of the ***virtual DOM*** and how changes in state affect changes in the actual DOM, let's implement the simple ***Counter*** component as shown below. A ***count*** variable is initialized and then rendered successfully on the screen. Buttons ***Up*** and ***Down*** successfully update the ***count*** variable as evidenced in the console, but the changes fail to update the DOM as desired. This happens because as far as React.js is concerned, there has been no changes to the virtual DOM, and therefore no need to update the actual DOM.

src/Labs/Lab4/Counter.tsx

```
import React, { useState } from "react";
export default function Counter() {
  let count = 7;
  console.log(count);
  return (
    <div id="wd-counter-use-state">
      <h2>Counter: {count}</h2>
      <button
        onClick={() => { count++; console.log(count); }}
        id="wd-counter-up-click">
        Up
      </button>
      <button
        onClick={() => { count--; console.log(count); }}
        id="wd-counter-down-click">
        Down
      </button>
<hr/></div>);}
```

```
// declare and initialize
// a variable. print changes
// of the variable to the console

// render variable

// variable updates on console
// but fails to update the DOM as desired
```



For the DOM to be updated as expected, we need to tell React.js that changes to a particular variable is indeed relevant to changes in the DOM. To do this, use the ***useState*** hook to declare the state variable, and update it using the mutator function as shown below. Now changes to the state variable are represented as changes in the DOM. Implement the

Counter component, import it in ***Lab4*** and confirm it works as expected. Do the same with the rest of the exercises that follow.

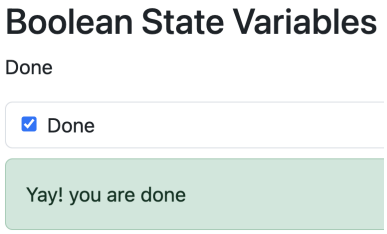
src/Labs/Lab4/Counter.tsx

```
import React, { useState } from "react";
export default function Counter() {
  let count = 7;
  const [count, setCount] = useState(7);
  console.log(count);
  return (
    <div>
      <h2>Counter: {count}</h2>
      <button onClick={() => setCount(count + 1)}
              id="wd-counter-up-click">Up</button>
      <button onClick={() => setCount(count - 1)}
              id="wd-counter-down-click">Down</button>
<hr/></div>);}
```

```
// import useState


// create and initialize
// state variable


// render state variable
// handle events and update
// state variable with mutator
// now updates to the state
// state variable do update the
// DOM as desired
```

2.3.3 Boolean State Variables

The ***useState*** hook works with all JavaScript data types and structures including ***booleans***, ***integers***, ***strings***, ***numbers***, ***arrays***, and ***objects***. The exercise below illustrates using the ***useState*** hook with ***boolean*** state variables. The variable is used to hide or show a DIV as well as render a checkbox as checked or not. Also note the use of ***onChange*** in the checkbox to set the value of state variable.



src/Labs/Lab4/BooleanStateVariables.tsx

```
import React, { useState } from "react";
export default function BooleanStateVariables() {
  const [done, setDone] = useState(true);
  return (
    <div id="wd-boolean-state-variables">
      <h2>Boolean State Variables</h2>
      <p>{done ? "Done" : "Not done"}</p>
      <label className="form-control">
        <input type="checkbox" checked={done}
               onChange={() => setDone(!done)} /> Done
      </label>
      {done && <div className="alert alert-success">
               Yay! you are done</div>}
<hr/></div>);}
```

```
// import useState

// declare and initialize
// boolean state variable


// render content based on
// boolean state variable value
// change state variable value
// when handling events like
// clicking a checkbox
// render content based on
// boolean state variable value
```

2.3.4 String State Variables

The ***StringStateVariables*** exercise below illustrates using ***useState*** with string state variables. The input field's ***value*** is initialized to the ***firstName*** state variable. The ***onChange*** attribute invokes the ***setFirstName*** mutator function to update the state variable. The ***e.target.value*** contains the value of the input field and is used to update the current value of the state variable.

src/Labs/Lab4/StringStateVariables.tsx

```
import React, { useState } from "react";
export default function StringStateVariables() {
  const [firstName, setFirstName] = useState("John");
  return (
    <div>
      <h2>String State Variables</h2>
      <p>{firstName}</p>
```

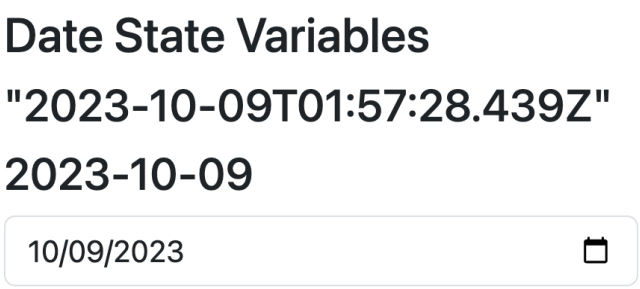
```
// import useState

// declare and
// initialize
// state variable

// render string
```

```
      <input
        className="form-control"
        defaultValue={firstName}
        onChange={(e) => setFirstName(e.target.value)}/>
<hr/></div>);}
```

```
// state variable
// initialize a
// text input field with the state variable
// update the state variable at each key stroke
```

2.3.5 Date State Variables

The ***DateStateVariable*** component illustrates how to work with date state variables. The ***stateDate*** state variable is initialized to the current date using ***new Date()*** which has the string representation as shown here on the right. The ***dateObjectToHtmlDateString*** function can convert a ***Date*** object into the ***YYYY-MM-DD*** format expected by the HTML date input field. The function is used to initialize and set the date field's ***value*** attribute so it matches the expected format. Changes in date field are handled by the ***onChange*** attribute which updates the new date using the ***setStartDate*** mutator function.



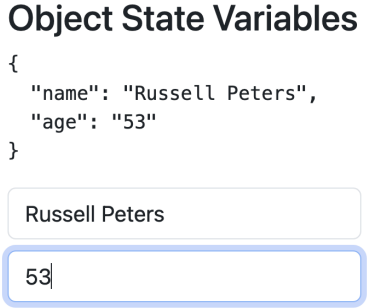
src/Labs/Lab4/DateStateVariable.tsx

```
import React, { useState } from "react";
export default function DateStateVariable() {
  const [startDate, setStartDate] = useState(new Date());
  const dateObjectToHtmlDateString = (date: Date) => {
    return `${date.getFullYear()}-${date.getMonth() + 1 < 10 ? 0 : ""}${
      date.getMonth() + 1
    }-${date.getDate() + 1 < 10 ? 0 : ""}${date.getDate() + 1}`;
  };
  return (
    <div id="wd-date-state-variables">
      <h2>Date State Variables</h2>
      <h3>{JSON.stringify(startDate)}</h3>
      <h3>{dateObjectToHtmlDateString(startDate)}</h3>
      <input
        className="form-control"
        type="date"
        defaultValue={dateObjectToHtmlDateString(startDate)}
        onChange={(e) => setStartDate(new Date(e.target.value))}
      />
<hr/></div>);}
```

```
// import useState

// declare and initialize with today's date
// utility function to convert date object
// to YYYY-MM-DD format for HTML date
// picker

// display raw date object
// display in YYYY-MM-DD format for input
// of type date

// set HTML input type date
// update when you change the date with
// the date picker
```

2.3.6 Object State Variables

The ***ObjectStateVariable*** component below demonstrates how to work with object state variables. We declare ***person*** object state variable with initial property values ***name*** and ***age***. The object is rendered on the screen using ***JSON.stringify*** to see the changes in real time. Two value of two input fields are initialized to the object's ***person.name*** string property and the object's ***person.age*** number property. As the user types in the input fields, the ***onChange*** attribute passes the events to update the object's property using the ***setPerson*** mutator functions. The object is updated by creating new objects copied from the previous object value using the spreader operator (***...person***), and then overriding the ***name*** or ***age*** property with the ***target.value***.



src/Labs/Lab4/ObjectStateVariable.tsx

```
import React, { useState } from "react";
export default function ObjectStateVariable() {
  const [person, setPerson] = useState({ name: "Peter", age: 24 });
  return (
    <div>
```

```
// import useState

// declare and initialize object state
// variable with multiple fields
```

```
      <h2>Object State Variables</h2>
      <pre>{JSON.stringify(person, null, 2)}</pre>
      <input
        defaultValue={person.name}
        onChange={(e) => setPerson({ ...person, name: e.target.value })}
      />
      <input
        defaultValue={person.age}
        onChange={(e) => setPerson({ ...person,
                                     age: parseInt(e.target.value) })}
      />
      <hr/>
    </div>
  );
}
```

```
// display raw JSON
// initialize input field with an object's
// field value
// update field as user types. copy old
// object, override specific field with new
// value
// update field as user types. copy old
// object,
// override specific field with new value
```

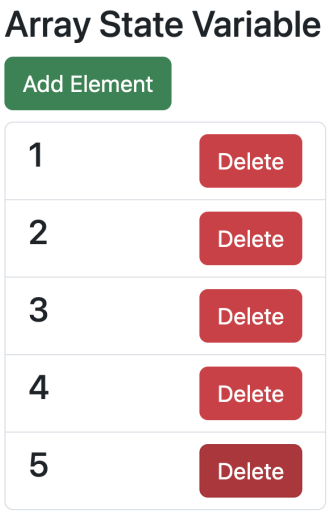
2.3.7 Array State Variables

The ***ArrayStateVariable*** component below demonstrates how to work with ***array*** state variables. An array of integers if declared as a state variable and function ***addElement*** and ***deleteElement*** are used to add and remove elements to and from the array. We render the array as a map of line items in an unordered list. We render the array's value and a ***Delete*** button for each element. Clicking the ***Delete*** button calls the ***deleteElement*** function which passes the ***index*** of the element we want to remove. The ***deleteElement*** function computes a new array filtering out the element by its position and updating the ***array*** state variable to contain a new array without the element we filtered out. Clicking the ***Add Element*** button invokes the ***addElement*** function which computes a new array with a copy of the previous ***array*** spread at the beginning of the new array, and adding a new random element at the end of the array. Add Bootstrap classes so the output renders as shown.

src/Labs/Lab4/ArrayStateVariable.tsx

```
import React, { useState } from "react";
export default function ArrayStateVariable() {
  const [array, setArray] = useState([1, 2, 3, 4, 5]);
  const addElement = () => {
    setArray([...array, Math.floor(Math.random() * 100)]);
  };
  const deleteElement = (index: number) => {
    setArray(array.filter((item, i) => i !== index));
  };
  return (
    <div id="wd-array-state-variables">
      <h2>Array State Variable</h2>
      <button onClick={addElement}>Add Element</button>
      <ul>
        {array.map((item, index) => (
          <li key={index}>
            {item}
            <button onClick={() => deleteElement(index)}
                    id="wd-delete-element-click">
              Delete</button>
          </li>
        ))}
      </ul>
      <hr/>
    </div>
  );
}
```

```
// import useState

// declare array state
// event handler appends
// random number at end of
// array
// event handler removes
// element by index

// button calls addElement
// to append to array
// iterate over array items

// render item's value
// button to delete element
// by its index
```



2.3.8 Sharing State Between Components

State can be shared between components by passing references to state variables and/or functions that update them. The example below demonstrates a ***ParentStateComponent*** sharing ***counter*** state variable and ***setCounter*** mutator function with ***ChildStateComponent*** by passing it references to ***counter*** and ***setCounter*** as attributes.

src/Labs/Lab4/ParentStateComponent.tsx

```
import React, { useState } from "react";
import ChildStateComponent from "./ChildStateComponent";
export default function ParentStateComponent() {
  const [counter, setCounter] = useState(123);
  return (
    <div>
      <h2>Counter {counter}</h2>
      <ChildStateComponent
        counter={counter}
        setCounter={setCounter} />
      <hr/>
    </div>
);}
```

The ***ChildStateComponent*** can use references to ***counter*** and ***setCounter*** to render the state variable and manipulate it through the mutator function. Import ***ParentStateComponent*** into ***Lab4*** and confirm it works as expected.

src/Labs/Lab4/ChildStateComponent.tsx

```
export default function ChildStateComponent({ counter, setCounter }:
{ counter: number;
  setCounter: (counter: number) => void;}) {
  return (
    <div id="wd-child-state">
      <h3>Counter {counter}</h3>
      <button onClick={() => setCounter(counter + 1)} id="wd-increment-child-state-click">
        Increment</button>
      <button onClick={() => setCounter(counter - 1)} id="wd-decrement-child-state-click">
        Decrement</button>
      <hr/>
    </div>
);}
```

2.4 Managing Application State

The ***useState*** hook is used to maintain the state within a component. State can be shared across components by passing references to state variables and mutators to other components. Although this approach is sufficient as a general approach to share state among multiple components, it is fraught with challenges when building larger, more complex applications. The downside of using ***useState*** across multiple components is that it creates an explicit dependency between these components, making it hard to refactor components adapting to changing requirements. The solution is to eliminate the dependency using libraries such as ***Redux***. This section explores the Redux library to manage state that is meant to be used across a large set of components, and even an entire application. We'll keep using ***useState*** to manage state within individual components, but use Redux to manage Application level state. To learn about redux, let's create a redux examples component that will contain several simple redux examples. Create an ***index.tsx*** file under ***src/Labs/Lab4/ReduxExamples/index.tsx*** as shown below. Import the new redux examples component into the Lab 4 component so we can see how it renders as we add new examples. Reload the browser and confirm the new component renders as expected.

src/Labs/Lab4/ReduxExamples/index.tsx

```
import React from "react";

export default function ReduxExamples() {
  return(
    <div>
      <h2>Redux Examples</h2>
    </div>
  );
};
```

src/Labs/Lab4/index.tsx

```
import React from "react";
import ReduxExamples from "./ReduxExamples";
export default const Lab4 = () => {
 return(
   <>
     <h2>Lab 4</h2>
     ...
     <ReduxExamples/>
   </>
 );};
```

2.4.1 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 --save
```

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 --save
npm  install  @reduxjs/toolkit --save
```

2.4.2 Create a Hello World Redux component

To learn about Redux, let's start with a simple Hello World example. Instead of maintaining state within any particular component, Redux declares and manages state in separate ***reducers*** which then ***provide*** the state to the entire application. Create ***helloReducer*** as shown below maintaining a state that consists of just a ***message*** state string initialized to ***Hello World***.

src/Labs/Lab4/ReduxExamples/HelloRedux/helloReducer.ts

```
import { createSlice } from "@reduxjs/toolkit";
const initialState = {
  message: "Hello World",
};
const helloSlice = createSlice({
  name: "hello",
  initialState,
  reducers: {},
});
export default helloSlice.reducer;
```

Application state can maintain data from various components or screens across an entire application. Each would have a separate reducer that can be combined into a single ***store*** where reducers come together to create a complex, application wide state. The ***store.tsx*** below demonstrates adding the ***helloReducer*** to the store. Later exercises and the ***Kanbas*** section will add additional reducers to the store.

src/Labs/store/index.ts

```
import { configureStore } from "@reduxjs/toolkit";
import helloReducer from "../Lab4/ReduxExamples/HelloRedux/helloReducer";
const store = configureStore({
  reducer: { helloReducer },
});
export default store;
```

The application state can then be shared with the entire Web application by wrapping it with a ***Provider*** component that makes the state data in the ***store*** available to all components within the ***Provider***'s body.

src/Labs/index.tsx

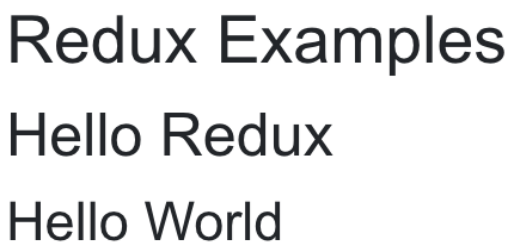
```
...
import store from "./store";
import { Provider } from "react-redux";
export default function Labs() {
  return (
    <Provider store={store}>
      <div className="container-fluid">
```

```
        <h1>Labs</h1>
        ...
      </div>
    </Provider>
  );
}
```

Components within the body of the ***Provider*** can then ***select*** the state data they want using the ***useSelector*** hook as shown below. Add the ***HelloRedux*** component to ***ReduxExamples*** and confirm it renders as shown below.

src/Labs/Lab4/ReduxExamples/HelloRedux/index.tsx

```
import { useSelector, useDispatch } from "react-redux";
export default function HelloRedux() {
  const { message } = useSelector((state: any) => state.helloReducer);
  return (
    <div id="wd-hello-redux">
      <h3>Hello Redux</h3>
      <h4>{message}</h4> <hr />
    </div>
  );
}
```



2.4.3 Counter Redux - Dispatching Events to Reducers

To practice with Redux, let's reimplement the ***Counter*** component using Redux. First create ***counterReducer*** responsible for maintaining the counter's state. Initialize the state variable ***count*** to 0, and reducer function ***increment*** and ***decrement*** can update the state variable by manipulating their ***state*** parameter that contain state variables as shown below.

src/Labs/Lab4/ReduxExamples/CounterRedux/counterReducer.tsx

```
import { createSlice } from "@reduxjs/toolkit";
const initialState = {
  count: 0,
};
const counterSlice = createSlice({
  name: "counter",
  initialState,
  reducers: {
    increment: (state) => {
      state.count = state.count + 1;
    },
    decrement: (state) => {
      state.count = state.count - 1;
    },
  },
});
export const { increment, decrement } = counterSlice.actions;
export default counterSlice.reducer;
```

Add the ***counterReducer*** to the ***store*** as shown below to make the counter's state available to all components within the body of the ***Provider***.

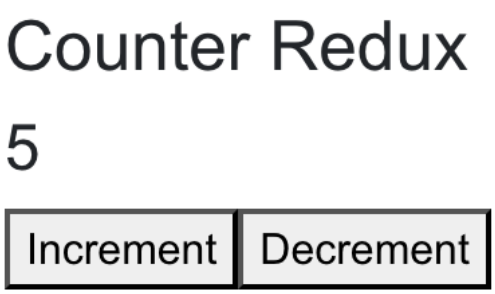
src/Labs/store/index.tsx

```
import { configureStore } from "@reduxjs/toolkit";
import helloReducer from "../Lab4/ReduxExamples/HelloRedux/helloReducer";
import counterReducer from "../Lab4/ReduxExamples/CounterRedux/counterReducer";
const store = configureStore({
  reducer: {
    helloReducer,
    counterReducer,
  },
});
export default store;
```

The ***CounterRedux*** component below can then ***select*** the ***count*** state from the store using the ***useSelector*** hook. To invoke the reducer function ***increment*** and ***decrement*** use a ***dispatch*** function obtained from a ***useDispatch*** function as shown below. Add ***CounterRedux*** to ***ReduxExamples*** and confirm it works as expected.

src/Labs/Lab4/ReduxExamples/CounterRedux/index.tsx

```
import { useSelector, useDispatch } from "react-redux";
import { increment, decrement } from "./counterReducer";
export default function CounterRedux() {
  const { count } = useSelector((state: any) => state.counterReducer);
  const dispatch = useDispatch();
  return (
    <div id="wd-counter-redux">
      <h2>Counter Redux</h2>
      <h3>{count}</h3>
      <button onClick={() => dispatch(increment())}
              id="wd-counter-redux-increment-click"> Increment </button>
      <button onClick={() => dispatch(decrement())}
              id="wd-counter-redux-decrement-click"> Decrement </button>
      <hr/>
    </div>
  );
}
```



2.4.4 Passing Data to Reducers

Now let's explore how the user interface can pass data to reducer functions. Create a reducer that can keep track of the arithmetic addition of two parameters. When we call ***add*** reducer function below, the parameters are encoded as an object into a ***payload*** property found in the ***action*** parameter passed to the reducer function. Functions can extract parameters ***a*** and ***b*** as ***action.payload.a*** and ***action.payload.b*** and then use the parameters to update the ***sum*** state variable.

src/Labs/Lab4/ReduxExamples/AddRedux/addReducer.tsx

```
import { createSlice } from "@reduxjs/toolkit";
const initialState = {
  sum: 0,
};
const addSlice = createSlice({
  name: "add",
  initialState,
  reducers: {
    add: (state, action) => {
      state.sum = action.payload.a + action.payload.b;
    },
  },
});
export const { add } = addSlice.actions;
export default addSlice.reducer;
```

Add the new reducer to the store so it's available throughout the application as shown below.

src/Labs/store/index.tsx

```
import { configureStore } from "@reduxjs/toolkit";
import helloReducer from "../Lab4/ReduxExamples/HelloRedux/helloReducer";
import counterReducer from "../Lab4/ReduxExamples/CounterRedux/counterReducer";
import addReducer from "../Lab4/ReduxExamples/AddRedux/addReducer";
const store = configureStore({
  reducer: {
    helloReducer, counterReducer,
    addReducer,
  },
});
export default store;
```

To tryout the new reducer, import the ***add*** reducer function as shown in the ***AddRedux*** component below. Maintain the values of ***a*** and ***b*** as local component state variables, and then pass them to ***add*** as a single object. Add ***AddRedux*** to ***ReduxExamples*** to confirm it works as expected.

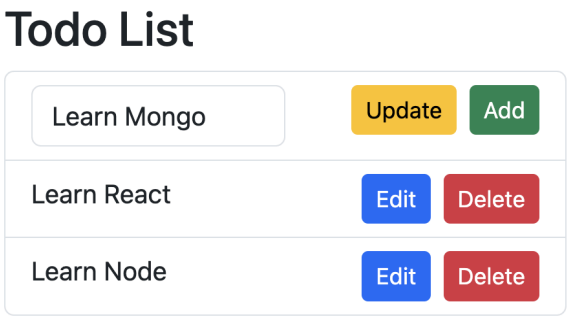
src/Labs/Lab4/ReduxExamples/AddRedux/index.tsx

```
import { useSelector, useDispatch } from "react-redux";
import { useState } from "react";
import { add } from "./addReducer";
export default function AddRedux() {
  const [a, setA] = useState(12);
  const [b, setB] = useState(23);
  const { sum } = useSelector((state: any) => state.addReducer);
  const dispatch = useDispatch();
  return (
    <div className="w-25" id="wd-add-redux">
      <h1>Add Redux</h1>
      <h2>{a} + {b} = {sum}</h2>
      <input type="number" defaultValue={a}
        onChange={(e) => setA(parseInt(e.target.value))}
        className="form-control" />
      <input type="number" defaultValue={b}
        onChange={(e) => setB(parseInt(e.target.value))}
        className="form-control" />
      <button className="btn btn-primary" id="wd-add-redux-click"
        onClick={() => dispatch(add({ a, b }))}>
        Add Redux </button>
      <hr/>
    </div>
  );
}
```

```
// to read/write to reducer
// to maintain a and b parameters in UI



// a and b state variables to edit
// parameters to add in the reducer
// read the sum state variable from the reducer
// dispatch to call add redux function

// render local state variables a and b, as well
// as application state variable sum

// update the local component state variable a

// update the local component state variable b


// on click, call add reducer function to
// compute the arithmetic addition of a and b,
// and store it in application state
// variable sum
```

2.5 Implementing a Todo List with Redux

Let's practice using local component state as well as application level state to implement a simple ***Todo List*** component. First we'll implement the component using only component state with ***useState*** which will limit the todos to only available within the ***Todo List***. We'll then add application state support to demonstrate how the todos can be shared with any component or screen in the application. Create the ***TodoList*** component as shown below. Add ***Bootstrap*** classes to style the todos as shown here on the right.



src/Labs/Lab4/ReduxExamples/todos/TodoList.tsx

```
import React, { useState } from "react";
export default function TodoList() {
  const [todos, setTodos] = useState([
    { id: "1", title: "Learn React" },
    { id: "2", title: "Learn Node"  }]);
  const [todo, setTodo] = useState({ id: "-1", title: "Learn Mongo" });
  const addTodo = (todo: any) => {
    const newTodos = [ ...todos, { ...todo,
      id: new Date().getTime().toString() }];
    setTodos(newTodos);
    setTodo({id: "-1", title: ""});
  };
  const deleteTodo = (id: string) => {
    const newTodos = todos.filter((todo) => todo.id !== id);
    setTodos(newTodos);
  };
  const updateTodo = (todo: any) => {
    const newTodos = todos.map((item) =>
      (item.id === todo.id ? todo : item));
    setTodos(newTodos);
    setTodo({id: "-1", title: ""});
  };
```

```
// import useState

// create todos array state variable
// initialize with 2 todo objects

// create todo state variable object
// event handler to add new todo
// spread existing todos, append new todo,
// override id
// update todos
// clear the todo

// event handler to remove todo by their ID



// event handler to
// update todo by
// replacing todo
// by their ID
```

```
  return (
    <div>
      <h2>Todo List</h2>
      <ul className="list-group">
        <li className="list-group-item">
          <button onClick={() => addTodo(todo)}
                  id="wd-add-todo-click">Add</button>
          <button onClick={() => updateTodo(todo)}
                  id="wd-update-todo-click">
            Update </button>
          <input defaultValue={todo.title}
            onChange={(e) =>
              setTodo({ ...todo,
                title: e.target.value })
            }
          />
        </li>
        {todos.map((todo) => (
          <li key={todo.id} className="list-group-item">
            <button onClick={() => deleteTodo(todo.id)}
                    id="wd-delete-todo-click">
              Delete </button>
            <button onClick={() => setTodo(todo)}
                    id="wd-set-todo-click">
              Edit </button>
            {todo.title}
          </li>
        ))}
      </ul>
      <hr/>
    </div>
  );
}
```

```
// add todo button
// update todo button

// input field to update todo's title

// for every keystroke
// update the todo's title,
// but copy old values first

// render all todos
// as line items
// button to delete todo by their ID

// button to select todo to edit
```

2.5.1 Breaking up Large Components

Let's break up the ***TodoList*** component into several smaller components: ***TodoItem*** and ***TodoForm***. ***TodoItem*** shown below breaks out the line items that render the todo's title, and ***Delete*** and ***Edit*** buttons. The component accepts references to the ***todo*** object, as well as ***deleteTodo*** and ***setTodo*** functions.

src/Labs/Lab4/ReduxExamples/todos/TodoItem.tsx

```
export default function TodoItem({ todo, deleteTodo, setTodo }: {
  todo: { id: string; title: string };
  deleteTodo: (id: string) => void;
  setTodo: (todo: { id: string; title: string }) => void;
}) {
  return (
    <li key={todo.id} className="list-group-item">
      <button onClick={() => deleteTodo(todo.id)}
              id="wd-delete-todo-click"> Delete </button>
      <button onClick={() => setTodo(todo)}
              id="wd-set-todo-click"> Edit </button>
      {todo.title}    </li>);}
```

```
// breaks out todo item
// todo to render
// event handler to remove todo
// event handler to select todo

// invoke delete todo with ID

// invoke select todo

// render todo's title
```

Similarly we'll break out the form to ***Create*** and ***Update*** todos into component ***TodoForm*** shown below. Parameters ***todo***, ***setTodo***, ***addTodo***, and ***updateTodo***, to maintain dependencies between the ***TodoList*** and ***TodoForm*** component.

src/Labs/Lab4/ReduxExamples/todos/TodoForm.tsx

```
export default function TodoForm({ todo, setTodo, addTodo, updateTodo }: {
  todo: { id: string; title: string };
  setTodo: (todo: { id: string; title: string }) => void;
  addTodo: (todo: { id: string; title: string }) => void;
  updateTodo: (todo: { id: string; title: string }) => void;
}) {
  return (
    <li className="list-group-item">
```

```
// breaks out todo form
// todo to be added or edited
// event handler to update todo's title
// event handler to add new todo
// event handler to update todo
```

```
      <button onClick={() => addTodo(todo)}
              id="wd-add-todo-click"> Add </button>
      <button onClick={() => updateTodo(todo)}
              id="wd-update-todo-click"> Update </button>
      <input defaultValue={todo.title}
        onChange={ (e) => setTodo({ ...todo, title: e.target.value }) }/>
    </li>
);}
```

```
// invoke add new todo
// invoke update todo
// input field to update
// todo's title
// update title on each key stroke
```

Now we can replace the form and todo items in the ***TodoList*** component as shown below. Add the ***TodoList*** component to ***Lab4*** and confirm it works as expected.

src/Labs/Lab4/ReduxExamples/todos/TodoList.tsx

```
import TodoForm from "./TodoForm";
import TodoItem from "./TodoItem";
export default function TodoList() {
  ...
  return (
    <div id="wd-todo-list-redux">
      <h2>Todo List</h2>
      <ul className="list-group">
        <TodoForm
          todo={todo}
          setTodo={setTodo}
          addTodo={addTodo}
          updateTodo={updateTodo}/>
        {todos.map((todo) => (
          <TodoItem
            todo={todo}
            deleteTodo={deleteTodo}
            setTodo={setTodo} />
        ))}
      </ul>
<hr/></div>);}
```

```
// import TodoForm
// import TotoItem

// TodoForm breaks out form to add or update todo
// pass state variables and
// event handlers
// so component
// can communicate with TodoList's data and functions

// TodoItem breaks out todo item
// pass state variables and
// event handlers to
// communicate with TodoList's data and functions
```

2.5.2 Todos Reducer

Although the ***TodoList*** component might work as expected and it might be all we would need, it's implementation makes it difficult to share the local state data (the todos) outside its context with other components or screens. For instance, how would we go about accessing and displaying the todos, say, in the ***Lab3*** component or ***Kanbas***? We would have to move the todos state variable and mutator functions to a component that is parent to both the ***Lab3*** component and the ***TodoList*** component, e.g., ***Labs*** or even ***App***.

Instead, let's move the state and functions from the ***TodoList*** component to a reducer and store so that the todos can be accessed from anywhere within the ***Labs***. Create ***todosReducer*** as shown below, moving the ***todos*** and ***todo*** state variables to the reducer's ***initialState***. Also move the ***addTodo***, ***deleteTodo***, ***updateTodo***, and ***setTodo*** functions into the ***reducers*** property, reimplementing them to use the ***state*** and ***action*** parameters of the new reducer functions.

src/Labs/Lab4/ReduxExamples/todos/todosReducer.ts

```
import { createSlice } from "@reduxjs/toolkit";
const initialState = {
  todos: [
    { id: "1", title: "Learn React" },
    { id: "2", title: "Learn Node" },
  ],
  todo: { title: "Learn Mongo" },
};
const todosSlice = createSlice({
  name: "todos",
  initialState,
  reducers: {
    addTodo: (state, action) => {
      const newTodos = [
```

```
// import createSlice
// declare initial state of reducer
// moved here from TodoList.tsx
// todos has default todos


// todo has default todo

// create slice
// name slice
// configure store's initial state
// declare reducer functions
// addTodo reducer function, action
// contains new todo. newTodos
```

```
        ...state.todos,
      { ...action.payload, id: new Date().getTime().toString() },
    ];
    state.todos = newTodos;
    state.todo = { title: "" };
  },
  deleteTodo: (state, action) => {
    const newTodos = state.todos.filter((todo) => todo.id !== action.payload);
    state.todos = newTodos;
  },
  updateTodo: (state, action) => {
    const newTodos = state.todos.map((item) =>
      item.id === action.payload.id ? action.payload : item
    );
    state.todos = newTodos;
    state.todo = { title: "" };
  },
  setTodo: (state, action) => {
    state.todo = action.payload;
  },
 },
});
export const { addTodo, deleteTodo, updateTodo, setTodo } = todosSlice.actions;
export default todosSlice.reducer;
```

```
// copy old todos, append new todo
// in action.payload, override
// id as timestamp
// update todos
// clear todo

// deleteTodo reducer function,
// action contains todo's ID to
// filter out of newTodos

// updateTodo reducer function
// rebuilding newTodos by replacing
// old todo with new todo in
// action.payload
// update todos
// clear todo

// setTodo reducer function
// to update todo state variable


// export reducer functions
// export reducer for store
```

Add the new ***todosReducer*** to the ***store*** so that it can be provided to the rest of the ***Labs***.

src/Labs/store/index.tsx

```
import { configureStore } from "@reduxjs/toolkit";
import helloReducer from "../Lab4/ReduxExamples/HelloRedux/helloReducer";
import counterReducer from "../Lab4/ReduxExamples/CounterRedux/counterReducer";
import addReducer from "../Lab4/ReduxExamples/AddRedux/addReducer";
import todosReducer from "../Lab4/ReduxExamples/todos/todosReducer";
const store = configureStore({
  reducer: {
    helloReducer,
    counterReducer,
    addReducer,
    todosReducer,
  },
});
export default store;
```

Now that we've moved the state and mutator functions to the ***todosReducer***, refactor the ***TodoForm*** component to use the reducer functions instead of the parameters. Also select the ***todo*** from the reducer state, instead of the ***todo*** parameter.

src/Labs/Lab4/ReduxExamples/todos/TodoForm.tsx

```
import React from "react";
import { useSelector, useDispatch } from "react-redux";
import { addTodo, updateTodo, setTodo } from "./todosReducer";

export default function TodoForm(
{ todo, setTodo, addTodo, updateTodo }
) {
  const { todo } = useSelector((state: any) => state.todosReducer);
  const dispatch = useDispatch();
  return (
    <li className="list-group-item">
      <button onClick={() => dispatch(addTodo(todo))}
              id="wd-add-todo-click"> Add </button>
      <button onClick={() => dispatch(updateTodo(todo))}
              id="wd-update-todo-click"> Update </button>
      <input
        defaultValue={todo.title}
        onChange={(e) => dispatch(setTodo({ ...todo, title: e.target.value }))}/>
    </li>
);}
```

```
// import useSelector, useDispatch
// to read/write to reducer
// reducer functions
// remove dependency from
// parent component

// retrieve todo from reducer
// create dispatch instance to
// invoke reducer functions

// wrap reducer functions
// with dispatch


// wrap reducer functions
// with dispatch
```

Also reimplement the ***TodoItem*** component as shown below, using the reducer functions instead of the parameters.

src/Labs/Lab4/ReduxExamples/todos/TodoItem.tsx

```
import React from "react";
import { useDispatch } from "react-redux";
import { deleteTodo, setTodo } from "./todosReducer";
export default function TodoItem({ todo,
  deleteTodo, setTodo
 }) {
  const dispatch = useDispatch();
  return (
    <li key={todo.id} className="list-group-item">
      <button onClick={() => dispatch(deleteTodo(todo.id))}
              id="wd-delete-todo-click"> Delete </button>
      <button onClick={() => dispatch(setTodo(todo))}
              id="wd-set-todo-click"> Edit </button>
      {todo.title}
    </li>
);}
```

```
// import useDispatch to invoke reducer
// functions deleteTodo and setTodo

// remove dependency with
// parent component
// create dispatch instance to invoke
// reducer functions

// wrap reducer functions with dispatch
```

Reimplement the ***TodoForm*** and ***TodoItem*** components as shown above and update the ***TodoList*** component as shown below. Remove unnecessary dependencies and confirm that it works as before.

src/Labs/Lab4/ReduxExamples/todos/TodoList.tsx

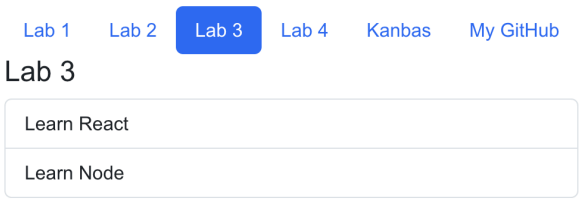
```
import React from "react";
import TodoForm from "./TodoForm";
import TodoItem from "./TodoItem";
import { useSelector } from "react-redux";
export default function TodoList() {
  const { todos } = useSelector((state: any) => state.todosReducer);
  return (
    <div id="wd-todo-list-redux">
      <h2>Todo List</h2>
      <ul className="list-group">
        <TodoForm />
        {todos.map((todo: any) => (
          <TodoItem todo={todo} />
        ))}
      </ul>
      <hr/>
    </div>
);}
```

```
// import useSelector to retrieve
// data from reducer

// extract todos from reducer and remove
// all other event handlers


// remove unnecessary attributes

// remove unnecessary attributes,
// but still pass the todo
```

Now the todos are available to any component in the body of the ***Provider***. To illustrate this, select the todos from within the ***Lab3*** component as shown below and confirm the todos display in ***Lab3***.

src/Labs/Lab3/index.tsx

```
...
import { useSelector } from "react-redux";

export default function Lab3() {
  const { todos } = useSelector((state: any) => state.todosReducer);
  return (
    <div>
      <h2>Lab 3</h2>
      <ul className="list-group">
        {todos.map((todo: any) => (
          <li className="list-group-item" key={todo.id}>
            {todo.title}
          </li>
        ))}
      </ul>
      <hr />
      ...
    </div>);}
```



3 Implementing the Kanbas User Interface

The current ***Kanbas*** implementation reads data from a ***Database*** containing ***courses***, ***modules***, ***assignments***, and ***grades***, and dynamically renders screens ***Dashboard***, ***Home***, ***Module***, ***Assignments***, and ***Grades***. The data is currently static, and our ***Kanbas*** implementation is basically a set of functions that transform the data in the ***Database*** into an corresponding user interface. Since the data is static, the user interface is static as well. In this section we will use the component and application state skills we learned in the ***Labs*** section, to refactor the ***Kanbas*** application so we can create new ***courses***, ***modules*** and ***assignments***.

3.1 Adding State to the Kanbas Dashboard

The current ***Dashboard*** implementation renders a static array of courses. Let's refactor the ***Dashboard*** to implement some ***CRUD*** operations such as ***create*** new courses, ***read*** courses, ***update*** existing course titles, and ***delete*** courses. Import the ***useState*** hook and convert the ***courses*** constant into a state variable as shown below. Make these changes in your current implementation using the code below as an example.

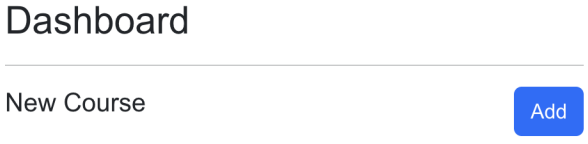
src/Kanbas/Dashboard.tsx

```
import React, { useState } from "react";
import { Link } from "react-router-dom";
import * as db from "../Database";
export default function Dashboard() {
  const [courses, setCourses] = useState<any[]>(db.courses);
  return (
    <div className="p-4" id="wd-dashboard">
      <h1 id="wd-dashboard-title">Dashboard</h1> <hr />
      <h2 id="wd-dashboard-published">Published Courses ({courses.length})</h2> <hr />
      <div className="row" id="wd-dashboard-courses">
        <div className="row row-cols-1 row-cols-md-5 g-4">
          {courses.map((course) => (
            <div key={course._id} className="col" style={{ width: "300px" }}>
              <div className="card">
                 ...  {course.name}   ...
              </div>
            </div>
          ))}
        </div>
      </div>
    </div>
);}
```

```
// add useState hook


// create courses state
// variable and initialize
// with database's courses
```

3.1.1 Creating New Courses

To create new courses, implement the ***addNewCourse*** function as shown below and add a new ***Add*** button that invokes ***addNewCourse*** function to append a new course at the end of the ***courses*** array. The ***addNewCourse*** function overrides the ***_id*** property with a unique timestamp.



src/Kanbas/Dashboard.tsx

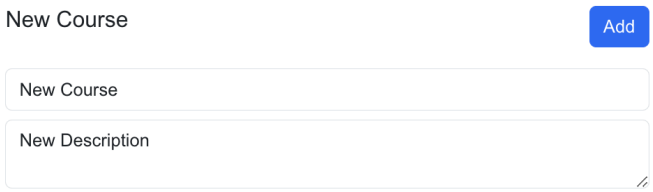
```
export default function Dashboard() {
  const [courses, setCourses] = useState<any[]>(db.courses);
  const course: any = {
    _id: "0", name: "New Course", number: "New Number",
    startDate: "2023-09-10", endDate: "2023-12-15",
    image: "/images/reactjs.jpg", description: "New Description"
```

```
// create a course object with default values
```

```
  };
  const addNewCourse = () => {
    const newCourse = { ...course,
                        _id: new Date().getTime().toString() };
    setCourses([...courses, { ...course, ...newCourse }]);
  };
  return (
    <div className="p-4" id="wd-dashboard">
      <h1 id="wd-dashboard-title">Dashboard</h1> <hr />
      <h5>New Course
          <button className="btn btn-primary float-end"
                  id="wd-add-new-course-click"
                  onClick={addNewCourse} > Add </button>
      </h5><hr />
      ...
    </div>
);}
```

```
// create addNewCourse event handler that sets
// courses as copy of current courses state array
// add course at the end of the array
// overriding _id to current time stamp


// add a title and button to invoke
// addNewCourse. Note no argument syntax
```

Confirm you can add new courses and the ***Published Courses*** counter increases. Modify the ***img*** tag so that it either renders a hardcoded image, e.g., ***"/images/react.jpg"***, or renders the course's ***image*** property, but then you'll need to add image properties to ***courses.json***. Convert the ***course*** constant into a state variable as shown below. Add a form to edit the ***course*** state variable's ***name***, and ***description*** properties. Confirm form shows values of the ***course*** state variable.



src/Kanbas/Dashboard.tsx

```
export default function Dashboard() {
  const [courses, setCourses] = useState<any[]>(db.courses);
  const [course, setCourse] = useState<any>({
    _id: "0", name: "New Course", number: "New Number",
    startDate: "2023-09-10", endDate: "2023-12-15",
    image: "/images/reactjs.jpg", description: "New Description"
  });
  const addNewCourse = () => { ... };
  return (
    <div id="wd-dashboard">
      <h1 id="wd-dashboard-title">Dashboard</h1> <hr />
      <h5>New Course ... </h5><br />
      <input defaultValue={course.name} className="form-control mb-2" />
      <textarea defaultValue={course.description} className="form-control"/>
      <hr />      ...
    </div>
);}
```

```
// convert course into a state
// variable so we can change it
// and force a redraw of the UI


// add input element for each of
// fields in course state
// variable
```

Add ***onChange*** attributes to each of the input fields to update each of the fields using the ***setCourse*** mutator function, as shown below. Use your implementation of ***Dashboard*** and the code provided as an example. Confirm you can add new courses.

src/Kanbas/Dashboard.tsx

```
  return (
    <div>
      <h1 id="wd-dashboard-title">Dashboard</h1> <hr />
      <h5>New Course ... </h5> <br />
      <input defaultValue={course.name} className="form-control mb-2"
             onChange={(e) => setCourse({ ...course, name: e.target.value }) } />
      <textarea defaultValue={course.description} className="form-control"
             onChange={(e) => setCourse({ ...course, description: e.target.value }) } />
      ...
    </div>
  );
```

3.1.2 Deleting a Course

Now let's implement deleting courses by adding ***Delete*** buttons to each of the courses. The buttons invoke a new ***deleteCourse*** function that accepts the ID of the course to remove. The function filters out the course from the ***courses*** array. Use the code below as an example to refactor your ***Dashboard*** component. Confirm that you can remove courses.

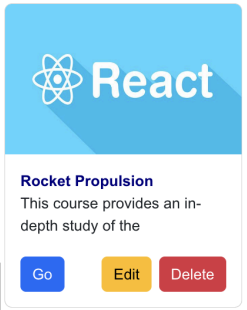
src/Kanbas/Dashboard.tsx

```
...
export default function Dashboard() {
  const [courses, setCourses] = useState<any[]>(db.courses);
  const [course, setCourse] = useState<any>({ ... });
  const addNewCourse = () => { ... };
  const deleteCourse = (courseId: string) => {
    setCourses(courses.filter((course) => course._id !== courseId));
  };
  return (
    <div id="wd-dashboard">
      <h1>Dashboard</h1>
      ...
      <div className="row">
        <div className="row row-cols-1 row-cols-md-5 g-4">
          {courses.map((course) => (
            ...
            <button className="btn btn-primary">
                    Go </button>
            <button onClick={(event) => {
                      event.preventDefault();
                      deleteCourse(course._id);
                    }} className="btn btn-danger float-end"
                    id="wd-delete-course-click">
                    Delete
            </button>
            ...
          ))}
        </div>
      </div>
    </div>
  );
}
```

```
// add deleteCourse event handler accepting
// ID of course to remove by filtering out
// the course by its ID
```

```
// add Delete button next to the course's
// name to invoke deleteCourse when clicked
// passing the course's ID and preventing
// the Link's default behavior to navigate
// to Course Screen
```

3.1.3 Editing a Course



Now let's implement editing an existing course by adding ***Edit*** buttons to each of the courses which invoke a new ***setCourse*** function that copies the current course into the ***course*** state variable, displaying the course in the form so you can edit it. Refactor your ***Dashboard*** component using the code below as an example. Confirm that clicking ***Edit*** on a course, copies the course into the form.

src/Kanbas/Dashboard.tsx

```
<button id="wd-edit-course-click"
  onClick={(event) => {
    event.preventDefault();
    setCourse(course);
  }}
  className="btn btn-warning me-2 float-end" >
  Edit
</button>
```

```
// next to the Delete button
// add an Edit button to copy the course
// to be edited into the form so we can
// edit it. Prevent default to navigate
// to Course screen
```

Add an ***Update*** button to the form so that the selected course can be updated with the values in the edited fields. Use the code below as an example. Confirm you can select, and then edit the selected course. Confirm that clicking ***Update*** actually updates the original course's ***name*** and ***description***.

src/Kanbas/Dashboard.tsx

```
...
export default function Dashboard() {
  const [courses, setCourses] = useState<any[]>(db.courses);
  const [course, setCourse] = useState<any>({ ... });
  const updateCourse = () => {
    setCourses(
      courses.map((c) => {
        if (c._id === course._id) {
          return course;
        } else {
          return c;
        }
      })
    );
  };
  return (
    <div id="wd-dashboard">
      <h1 id="wd-dashboard-title">Dashboard</h1>
      <hr />
      <h5>
        New Course
        <button className="btn btn-primary float-end"
                onClick={addNewCourse} id="wd-add-new-course-click">
          Add
        </button>
        <button className="btn btn-warning float-end me-2"
                onClick={updateCourse} id="wd-update-course-click">
          Update
        </button>
      </h5>
      ...
    </div>
  );
}
```

3.2 Courses Screen

The ***Dashboard*** component seems to be working fine, but the courses it is creating, deleting, and updating can not be used outside of the component. This is a problem because the ***Courses*** screen would want to be able to render the new courses, but it doesn't have access to the ***courses*** state variable in the ***Dashboard***. To fix this we need to either add redux so all courses are available everywhere, or move the ***courses*** state variable up to a component that is parent to both the ***Dashboard*** and the ***Courses***. Let's take this last approach first, and then we'll explore adding ***Redux***. Let's move all the state variables and event handlers from the ***Dashboard***, and move them to the ***Kanbas*** component since it is parent to both the ***Dashboard*** and ***Courses*** component. Then add references to the state variables and event handlers as parameter dependencies in ***Dashboard*** as shown below. Refactor your ***Dashboard*** component based on the example code below.

src/Kanbas/Dashboard.tsx

```
...
export default function Dashboard(
{ courses, course, setCourse, addNewCourse,
  deleteCourse, updateCourse }: {
  courses: any[]; course: any; setCourse: (course: any) => void;
  addNewCourse: () => void; deleteCourse: (course: any) => void;
  updateCourse: () => void; })
 {
  return (
    <div id="wd-dashboard">
      <h1>Dashboard</h1>
      ...
    </div>
  ); }
```

```
// move the state variables and
// event handler functions
// to Kanbas and then accept
// them as parameters
```

Refactor your ***Kanbas*** component moving the state variables and functions from the ***Dashboard*** component. Confirm the ***Dashboard*** still works the same, e.g., renders the courses, can add, updates, and remove courses

src/Kanbas/index.tsx

```
import KanbasNavigation from "./KanbasNavigation";
import { Routes, Route, Navigate } from "react-router-dom";
import Dashboard from "./Dashboard";
import Courses from "./Courses";
import * as db from "./Database";                                  // import the database
import { useState } from "react";                                  // import the useState hook

export default function Kanbas() {
  const [courses, setCourses] = useState<any[]>(db.courses);       // move the state variables here
  const [course, setCourse] = useState<any>({                      // from the Dashboard
    _id: "1234", name: "New Course", number: "New Number",
    startDate: "2023-09-10", endDate: "2023-12-15", description: "New Description",
  });
  const addNewCourse = () => {                                     // move the event handlers here
    setCourses([...courses, { ...course, _id: new Date().getTime().toString() }]);  // from the Dashboard
  };
  const deleteCourse = (courseId: any) => {
    setCourses(courses.filter((course) => course._id !== courseId));
  };
  const updateCourse = () => {
    setCourses(
      courses.map((c) => {
        if (c._id === course._id) {
          return course;
        } else {
          return c;
        }
      })
    );
  };
  return (
    <div id="wd-kanbas">
      <KanbasNavigation />
      <div className="wd-main-content-offset p-3">
        <Routes>
          <Route path="/" element={<Navigate to="Dashboard" />} />
          <Route path="Account" element={<h1>Account</h1>} />
          <Route path="Dashboard" element={
            <Dashboard                                             // pass a reference of the state
              courses={courses}                                    // variables and event handlers to
              course={course}                                      // the Dashboard so it can read
              setCourse={setCourse}                                // the state variables and invoke
              addNewCourse={addNewCourse}                          // the event handlers from the
              deleteCourse={deleteCourse}                          // Dashboard
              updateCourse={updateCourse}/>
          } />
          <Route path="Courses/:cid/*" element={<Courses courses={courses} />} />  // also pass all the courses to
        </Routes>                                                  // the Courses screen since now
      </div>                                                       // it might contain new courses
    </div>);}                                                      // not initially in the database
```

Now that we have the ***courses*** declared in the ***Kanbas*** component, we can share them with the ***Courses*** screen component by passing them as an attribute. The ***Courses*** component destructs the courses from the parameter and then finds the course by the ***courseId*** path parameter searching through the ***courses*** parameter instead of the ***courses*** in the ***Database***. Refactor your ***Courses*** component as suggested below and confirm you can navigate to new courses created in the ***Dashboard***.

src/Kanbas/Courses/index.tsx

```
// import { courses } from "../Database";
export default function Courses({ courses }: { courses: any[]; }) {   // don't load courses from Database
  const { cid } = useParams();                                        // accept courses from Kanbas
  const course = courses.find((course) => course._id === cid);
  return (...);                                                       // find the course by its ID
}
```

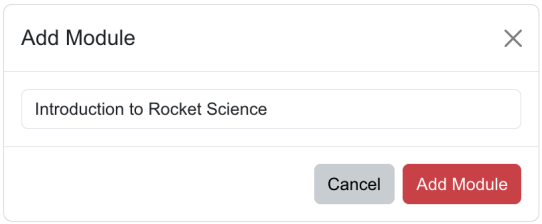
3.3 Modules

Now let's do the same with ***Modules*** refactoring the component adding state variables so that we can create, update, and remove modules. We'll discover the same limitation we had with ***courses***, i.e., we won't be able to share new modules outside the ***Modules*** screen. But instead of moving the modules state variable and functions to a shared common parent component, we'll instead use ***Redux*** to make the modules available throughout the application. The screenshot here on the right is for illustration purposes only. Reuse the HTML and CSS from previous assignments to style your modules. Refactor your ***Modules*** implementation by converting the ***modules*** array into a state variable as shown below. Confirm ***Modules*** renders as expected. Styling shown here is for illustration purposes. Use your HTML and CSS from previous assignments to style the modules.

src/Kanbas/Courses/Modules/index.tsx

```
import React, { useState } from "react";
import { useParams } from "react-router";
import * as db from "../../Database";
export default function Modules() {
  const { cid } = useParams();
  const [modules, setModules] = useState<any[]>(db.modules);
  return ( ... );
}
```

3.3.1 Creating a Module



Let's create a modal dialog where users can edit the name of a new module based on ***Bootstrap***'s modal classes. The ***ModuleEditor*** component below pops up when you click the red ***+ Module*** button in the ***Modules*** and ***Home*** screens. You can type the name of the new module in an input field. As you type the name of the module, the ***setModuleName*** updates the module name, and clicking on the ***Add Module*** button calls the ***addModule*** function which actually adds the module.

src/Kanbas/Courses/Modules/ModuleEditor.tsx

```
export default function ModuleEditor({ dialogTitle, moduleName, setModuleName, addModule }:
{ dialogTitle: string; moduleName: string; setModuleName: (name: string) => void; addModule: () => void; }) {
  return (
    <div id="wd-add-module-dialog" className="modal fade" data-bs-backdrop="static" data-bs-keyboard="false">
      <div className="modal-dialog">
        <div className="modal-content">
          <div className="modal-header">
            <h1 className="modal-title fs-5" id="staticBackdropLabel">
              {dialogTitle} </h1>
            <button type="button" className="btn-close" data-bs-dismiss="modal"></button>
          </div>
          <div className="modal-body">
            <input className="form-control" defaultValue={moduleName} placeholder="Module Name"
                   onChange={(e) => setModuleName(e.target.value)}/>
          </div>
          <div className="modal-footer">
            <button type="button" className="btn btn-secondary" data-bs-dismiss="modal">
              Cancel </button>
            <button onClick={addModule} type="button" data-bs-dismiss="modal" className="btn btn-danger">
              Add Module </button>
          </div>
        </div>
      </div>
    </div>
  );
}
```

The ***+ Module*** button was implemented in the ***ModulesControls*** in a prior assignment. Let's refactor it so that it displays the ***ModuleEditor*** dialog when clicked.

src/Kanbas/Courses/Modules/ModulesControls.tsx

```
import ModuleEditor from "./ModuleEditor";
export default function ModulesControls(
{ moduleName, setModuleName, addModule }:
{ moduleName: string; setModuleName: (title: string) => void; addModule: () => void; }) {
  return (
    <div id="wd-modules-controls" className="text-nowrap">
      <button className="btn btn-lg btn-danger me-1 float-end" id="wd-add-module-btn"
        data-bs-toggle="modal" data-bs-target="#wd-add-module-dialog" >
        <FaPlus className="position-relative me-2" style={{ bottom: "1px" }} />
        Module
      </button>
      ...
      <ModuleEditor dialogTitle="Add Module" moduleName={moduleName}
                    setModuleName={setModuleName} addModule={addModule} />
    </div>
);}
```

In the ***Modules*** screen, declare a ***moduleName*** state variable that keeps track of the module name edited in the ***ModuleEditor*** dialog. The ***addModule*** function below creates a new module instance with the ***moduleName*** as the name and appends it to the end of the ***modules*** state variable. The ***setModuleName*** and ***addModule*** functions are passed to the ***ModulesControls*** component which will in turn pass them to the ***ModuleEditor*** dialog. The ***ModuleEditor*** dialog will invoke the ***setModuleName*** function when editing the module name in the dialog text field. The dialog will invoke the ***addModule*** function when the ***Add Module*** button is clicked. Confirm you can add modules.

src/Kanbas/Courses/Modules/index.tsx

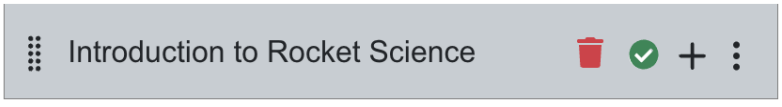
```
export default function Modules() {
  const { cid } = useParams();
  const [modules, setModules] = useState<any[]>(db.modules);
  const [moduleName, setModuleName] = useState("");
  const addModule = () => {
    setModules([ ...modules, { _id: new Date().getTime().toString(),
                               name: moduleName, course: cid, lessons: [] } ]);
    setModuleName("");
  };
  return (
    <div className="wd-modules">
      <ModulesControls setModuleName={setModuleName} moduleName={moduleName} addModule={addModule} />
      ...
    </div>
);}
```

3.3.2 Deleting a Module

To delete modules, let's add a ***trashcan*** icon to the ***ModuleControlButtons*** implement in an earlier assignment. We'll pass it a ***deleteModule*** we can call when clicking the ***trashcan*** and also pass the ID of the module to be deleted ***moduleId***.

src/Kanbas/Courses/Modules/ModuleControlButtons.tsx

```
import { FaTrash } from "react-icons/fa";
export default function ModuleControlButtons(
{ moduleId, deleteModule }: { moduleId: string; deleteModule: (moduleId: string) => void; } ) {
  return (
    <div className="float-end">
      <FaTrash className="text-danger me-2 mb-1" onClick={() => deleteModule(moduleId)}/>
      <GreenCheckmark />
      <BsPlus className="fs-1" />
      <IoEllipsisVertical className="fs-4" />
    </div>
);}
```



In the ***Modules*** screen, let's implement ***deleteModule*** function that removes a module by its ***ID*** and then pass the function to the ***ModuleControlButtons*** component along ***moduleId*** to be removed. Confirm you can remove modules.

src/Kanbas/Courses/Modules/index.tsx

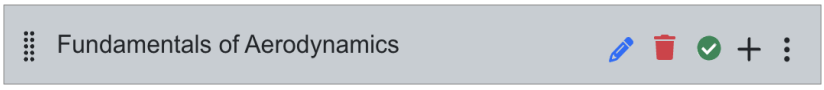
```
export default function Modules() {
  const addModule = () => { ... };
  const deleteModule = (moduleId: string) => {
    setModules(modules.filter((m) => m._id !== moduleId));
  };
  return (
    <div className="wd-modules">
      ...
      <ModuleControlButtons
        moduleId={module._id}
        deleteModule={deleteModule}/>
      ...
    </div>
  );
}
```

3.3.3 Editing a Module

In ***ModuleControlButtons***, add a pencil icon as shown below. Clicking the icon should call ***editModule*** with the ***moduleId*** of the module we want to edit.

src/Kanbas/Courses/Modules/ModuleControlButtons.tsx

```
import { FaTrash } from "react-icons/fa";
import { FaPencil } from "react-icons/fa6";
export default function ModuleControlButtons({ moduleId, deleteModule, editModule }: {
  moduleId: string; deleteModule: (moduleId: string) => void;
  editModule: (moduleId: string) => void }) {
  return (
    <div className="float-end">
      <FaPencil onClick={() => editModule(moduleId)} className="text-primary me-3" />
      <FaTrash .../>
      <GreenCheckmark />
      <BsPlus className="fs-1" />
      <IoEllipsisVertical className="fs-4" />
    </div>
);}
```



In the ***Modules*** component, implement functions ***editModule*** and ***updateModule*** as shown below. Pass the ***editModule*** function to the ***ModuleControlsButtons*** component so that when the ***pencil*** icon is clicked, it will invoke the ***editModule*** to set the module's ***editing*** field to true. The ***updateModule*** accepts a ***module*** object and updates the corresponding ***module*** object in the ***modules*** array. If the ***module***'s ***editing*** field is not set (false), then the ***module***'s ***name*** is displayed. But if the ***pencil*** is clicked, the ***module***'s ***editing*** field is set to true and instead of the ***module***'s ***name***, an input field is displayed so the ***name*** can be edited using the ***updateModule*** function. If the ***Enter*** key is pressed, the ***module***'s ***editing*** field is set to false, and then the editing field is hidden and the ***module***'s name is shown again. Confirm you can edit the names of the modules.

src/Kanbas/Courses/Modules/index.tsx

```
export default function Modules() {
  const { cid } = useParams();
  const [modules, setModules] = useState<any[]>(db.modules);
  const [moduleName, setModuleName] = useState("");
  const addModule = () => { ... }
  const deleteModule = (moduleId: string) => { ... }
  const editModule = (moduleId: string) => {
    setModules(modules.map((m) => (m._id === moduleId ? { ...m, editing: true } : m)));
```

```
// set the module's editing flag
// to true so that we can display
```

```
  };
  const updateModule = (module: any) => {
    setModules(modules.map((m) => (m._id === module._id ? module : m)));
  };
  return (
    <div className="wd-title p-3 ps-2 bg-secondary">
      <BsGripVertical className="me-2 fs-3" />
      {!module.editing && module.name}
      { module.editing && (
        <input className="form-control w-50 d-inline-block"
               onChange={(e) => updateModule({ ...module, name: e.target.value })}
               onKeyDown={(e) => {
                 if (e.key === "Enter") {
                   updateModule({ ...module, editing: false });
                 }
               }}
               defaultValue={module.name}/>
      )}
      <ModuleControlButtons
        moduleId={module._id}
        deleteModule={deleteModule}
        editModule={editModule}/>
      </div>
    </div>
  );
}
```

```
// the input field to edit name
// update any field(s) of a
// module

// show name if not editing
// show input field if editing
// when typing edit the module's
// name
// if "Enter" key is
// pressed then set editing
// field to false so we
// hide the text field

// pass editModule function to
// so if pencil is clicked we can
// set editing to true
```

3.3.4 Module Reducer

The ***Modules*** component seems to be working fine. We can create new modules, edit modules, and remove modules, BUT, it suffers a major flaw. Those new modules and edits can't be used outside the confines of the ***Modules*** component even though we would want to display the same list of modules elsewhere such as the ***Home*** screen. We could use the same approach as we did for the ***Dashboard***, by moving the state variables and functions to a higher level component that could share the state with other components. Instead we're going to use ***Redux*** this time to practice application level state management. To start, create the ***reducer.tsx*** shown below containing the ***modules*** state variables as well as the ***addModule***, ***deleteModule***, ***updateModule***, and ***editModule*** functions reimplemented in the ***reducers*** property.

src/Kanbas/Courses/Modules/reducer.ts

```
import { createSlice } from "@reduxjs/toolkit";
import { modules } from "../../Database";
const initialState = {
  modules: modules,
};
const modulesSlice = createSlice({
  name: "modules",
  initialState,
  reducers: {
    addModule: (state, { payload: module }) => {
      const newModule: any = {
        _id: new Date().getTime().toString(),
        lessons: [],
        name: module.name,
        course: module.course,
      };
      state.modules = [...state.modules, newModule] as any;
    },
    deleteModule: (state, { payload: moduleId }) => {
      state.modules = state.modules.filter(
        (m: any) => m._id !== moduleId);
    },
    updateModule: (state, { payload: module }) => {
      state.modules = state.modules.map((m: any) =>
        m._id === module._id ? module : m
      ) as any;
    },
    editModule: (state, { payload: moduleId }) => {
      state.modules = state.modules.map((m: any) =>
        m._id === moduleId ? { ...m, editing: true } : m
```

```
// import createSlice
// import modules from database
// create reducer's initial state with
// default modules copied from database

// create slice
// name the slice
// set initial state
// declare reducer functions
// new module is in action.payload
// update modules in state adding new module
// at beginning of array. Override _id with
// timestamp

// module's ID to delete is in action.payload
// filter out module to delete

// module to update is in action.payload
// replace module whose ID matches
// action.payload._id

// select the module to edit
```

```
      ) as any;
    },
  },
});
export const { addModule, deleteModule, updateModule, editModule } =
  modulesSlice.actions;
export default modulesSlice.reducer;
```

```
// export all reducer functions

// export reducer
```

The reducers, ***store***, and ***Provider*** we worked on for the ***Labs*** only wrapped the lab exercises, so those won't be available here in ***Kanbas***. Instead, let's create a new ***store*** and ***Provider*** specific for the ***Kanbas*** application. Create a new store as shown below.

src/Kanbas/store.ts

```
import { configureStore } from "@reduxjs/toolkit";
import modulesReducer from "./Courses/Modules/reducer";
const store = configureStore({
  reducer: {
    modulesReducer,
  },
});
export default store;
```

Then provide the store to the whole ***Kanbas*** application as shown below.

src/Kanbas/index.tsx

```
import store from "./store";
import { Provider } from "react-redux";
export default function Kanbas() {
  return (
    <Provider store={store}>
      <div id="wd-kanbas">
      ...
      </div>
    </Provider>
  );
}
```

```
// import the redux store
// import the redux store Provider


// wrap your application with the Provider so all
// child elements can read and write to the store
```

Reimplement the ***Modules*** by removing the state variables and functions, and replacing them with selectors, dispatchers, and reducer functions as shown below. Confirm you can still add, remove, and edit modules as before. Also confirm the modules still work in the ***Home*** screen.

src/Kanbas/Courses/Modules/index.tsx

```
import { addModule, editModule, updateModule, deleteModule }
  from "./reducer";
import { useSelector, useDispatch } from "react-redux";
export default function Modules() {
  const { cid } = useParams();
  const [moduleName, setModuleName] = useState("");
  const { modules } = useSelector((state: any) => state.modulesReducer);
  const dispatch = useDispatch();
  return (
    <div className="wd-modules">
      <ModulesControls moduleName={moduleName} setModuleName={setModuleName}
        addModule={() => {
          dispatch(addModule({ name: moduleName, course: cid }));
          setModuleName("");
        }} />
      <ul id="wd-modules" className="list-group rounded-0">
        {modules
          .filter((module: any) => module.course === cid)
          .map((module: any) => (
                {!module.editing && module.name}
                { module.editing && (
                  <input className="form-control w-50 d-inline-block"
```

```
// import reducer functions to add,
// delete, and update modules
// import useSelector and useDispatch


// retrieve modules state variables
// get dispatch to call reducer
// functions


// wrap reducer functions with
// dispatch clear module name
```

```
                    onChange={(e) =>
                      dispatch(
                        updateModule({ ...module, name: e.target.value })
                      )
                    }
                    onKeyDown={(e) => {
                      if (e.key === "Enter") {
                        dispatch(updateModule({ ...module, editing: false }));
                      }
                    }}
                    defaultValue={module.name} />
                )}
                <ModuleControlButtons moduleId={module._id}
                  deleteModule={(moduleId) => {
                    dispatch(deleteModule(moduleId));
                  }}
                  editModule={(moduleId) => dispatch(editModule(moduleId))} />
        ...
      </ul>
    </div>
  );
}
```

```
// wrap reducer functions with
// dispatch

// wrap reducer functions with
// dispatch

// wrap reducer
// functions with
// dispatch
```

3.4 Account Screens

The ***Account Screens*** provide users access to their personal information and all related data such as courses they are enrolled in and courses they might be teaching. Users use the ***Sign In*** screen to identify themselves and access their ***Profile*** screen to view their personal information. This section describes refactoring the ***Signin*** and ***Profile*** screens to confirm a user's identity and display their personal information.

3.4.1 Account Reducer

Implement an ***account reducer*** to keep track of the currently signed in user and share it across the entire application. Implement the ***account reducer*** as shown below and then add it to the ***Kanbas*** store.

src/Kanbas/Account/reducer.ts

```
import { createSlice } from "@reduxjs/toolkit";
const initialState = {
  currentUser: null,
};
const accountSlice = createSlice({
  name: "account",
  initialState,
  reducers: {
    setCurrentUser: (state, action) => {
      state.currentUser = action.payload;
    },
  },
});
export const { setCurrentUser } = accountSlice.actions;
export default accountSlice.reducer;
```

src/Kanbas/store.ts

```
import { configureStore } from "@reduxjs/toolkit";
import modulesReducer from "./Courses/Modules/reducer";
import accountReducer from "./Account/reducer";
const store = configureStore({
  reducer: {
    modulesReducer,
    accountReducer,
  },
});
export default store;
```

3.4.2 Signin

Refactor the ***Signin*** screen by adding a ***credentials*** state variable for users to enter their credentials. When users click the ***Sign In*** button, search for a user with the credentials. If there's a user that matches, store it in the reducer by ***dispatch***ing it to the ***Account reducer*** using the ***setCurrentUser*** reducer function. Ignore the ***Sign In*** attempt if there's no match. After signing in, navigate to the ***Dashboard*** as shown below. Confirm that signing in navigates to the ***Dashboard*** only if valid credentials are used.

src/Kanbas/Account/Signin.tsx

```
import { useState } from "react";
import { Link, useNavigate } from "react-router-dom";
import { setCurrentUser } from "./reducer";
import { useDispatch } from "react-redux";
import * as db from "../Database";

export default function Signin() {
  const [credentials, setCredentials] = useState<any>({});
  const dispatch = useDispatch();
  const navigate = useNavigate();
  const signin = () => {
    const user = db.users.find(
      (u: any) => u.username === credentials.username && u.password === credentials.password);
    if (!user) return;
    dispatch(setCurrentUser(user));
    navigate("/Kanbas/Dashboard");
  };
  return (
    <div id="wd-signin-screen">
      <h1>Sign in</h1>
      <input defaultValue={credentials.username}
             onChange={(e) => setCredentials({ ...credentials, username: e.target.value })}
             className="form-control mb-2" placeholder="username" id="wd-username" />
      <input defaultValue={credentials.password}
             onChange={(e) => setCredentials({ ...credentials, password: e.target.value })}
             className="form-control mb-2" placeholder="password" type="password" id="wd-password" />
      <button onClick={signin} id="wd-signin-btn" className="btn btn-primary w-100" > Sign in </button>
      <Link id="wd-signup-link" to="/Kanbas/Account/Signup"> Sign up </Link>
    </div>
);}
```

3.4.3 Dashboard

Now that the current user is stored in the ***Account reducer***, the ***Dashboard*** can filter the courses and only display the courses in which the current user is enrolled in. Refactor the ***Dashboard*** so that it only shows the courses the current user is enrolled in as shown below. Sign in as different users and confirm that the ***Dashboard*** only displays the courses a user is enrolled in. Note that new courses added will not render now since enrollments would also need to be modified. This will be addressed in later assignments.

src/Kanbas/Dashboard.tsx

```
import { useSelector } from "react-redux";
import * as db from "./Database";
...
export default function Dashboard(...) {
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  const { enrollments } = db;
  return(
    ...
  {courses
    .filter((course) =>
      enrollments.some(
        (enrollment) =>
          enrollment.user === currentUser._id &&
          enrollment.course === course._id
        ))
    .map((course) => (
      <div className="wd-dashboard-course col" style={{ width: "300px" }} >
        ...
      </div>
  ))}
   ...
);}
```

3.4.4 Protecting Routes and Content

Now the ***Dashboard*** depends on a user being signed, the screen should only be accessible if the a user has signed in. Navigation to the ***Dashboard*** needs to be ***protected*** from users that are not signed in. Often applications have screens that are only accessible if users are logged in, usually because the information is sensitive and/or the information they are accessing is based on the identify of the user. We can protect navigating to certain routes in the user interface by checking if a user is signed in already or not and then either allowing access to the route, or navigating users to the sign in screen instead. The ***ProtectedRoute*** component below uses the ***currentUser*** in the store to determine whether there's someone signed in or not. The ***children*** parameter is a reference to the protected screen or component and if there's someone signed in, the ***ProtectedRoute*** returns the ***children*** reference allowing the signed in user to access the route. If no one is signed in, ***ProtedRoute*** navigates the user to the ***Sign in*** screen.

src/Kanbas/Account/ProtectedRoute.tsx

```
import { useSelector } from "react-redux";
import { Navigate } from "react-router-dom";
export default function ProtectedRoute({ children }: { children: any }) {
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  if (currentUser) {
    return children;
  } else {
    return <Navigate to="/Kanbas/Account/Signin" />;
}}
```

Use ***ProtectedRoute*** to protect the ***Dashboard*** and ***Courses*** routes so that users will only be able to navigate there if they are signed in. Confirm that the ***Dashboard*** and ***Courses*** screens are only accessible if the users are signed in.

src/Kanbas/index.tsx

```
<Routes>
  <Route path="/" element={<Navigate to="Dashboard" />} />
  <Route path="Account/*"       element={<Account />} />
  <Route path="Dashboard"       element={<ProtectedRoute><Dashboard ...             /></ProtectedRoute> } />
  <Route path="Courses/:cid/*" element={<ProtectedRoute><Courses courses={courses} /></ProtectedRoute> } />
  <Route path="Calendar"        element={<h1>Calendar</h1>} />
  <Route path="Inbox"           element={<h1>Inbox</h1>} />
</Routes>
```

On your own, use the the current user's role to only allow ***FACULTY*** to edit any content such as courses, modules, and assignments. If a user does not have the ***FACULTY*** role, hide all forms and buttons that would allow editing any content, e.g., ***New Course*** form, ***Add***, ***Delete***, ***Edit*** and ***Update*** buttons for ***Courses***, ***Modules***, and ***Assignments***, etc.

3.4.5 Account Navigation

Users can use the ***Account Navigation*** sidebar to navigate between the ***Account Screen***s ***Signin***, ***Signup***, and ***Profile***, but not all screens should be available if there's a current user or not. Reimplement the ***Account Navigation*** sidebar so that it hides the ***Signin*** and ***Signup*** navigation links if a user is already signed in, and hides the ***Profile*** link if a user is not yet signed in.

src/Kanbas/Account/Navigation.tsx

```
import { Link, useLocation } from "react-router-dom";
import { useSelector } from "react-redux";
export default function AccountNavigation() {
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  const links = currentUser ? ["Profile"] : ["Signin", "Signup"];
  const { pathname } = useLocation();
  ...
}
```

Also refactor the ***Account*** screen so that the default screen is ***Signin*** if no one is signed in yet, and ***Profile*** if someone is already signed in.

src/Kanbas/Account/index.tsx

```
import { useSelector } from "react-redux";
export default function Account() {
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  ...
    <Routes>
      <Route path="/" element={<Navigate to={ currentUser ? "/Kanbas/Account/Profile" : "/Kanbas/Account/Signin" }/>}/>
      <Route path="/Signin" element={<Signin />} />
      <Route path="/Signup" element={<Signup />} />
      <Route path="/Profile" element={<Profile />} />
    </Routes>
  ...
);}
```

Confirm that the ***Account Navigation*** links are ***Sign In*** and ***Sign Up*** if no one is signed in yet, and ***Profile*** if someone is already signed in. Also confirm that clicking the ***Account*** link in the ***Kanbas Navigation*** sidebar displays the ***Signin*** screen if no one is signed in yet, and displays the ***Profile*** screen if someone is already signed in.

3.4.6 Profile

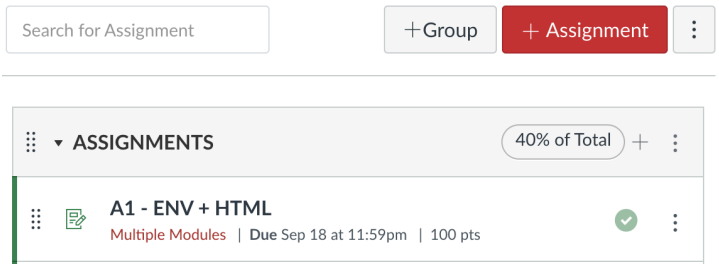
The ***Profile*** screen displays the current user's personal information. Refactor the ***Profile*** screen to retrieve the current user from the ***Account reducer***. If there's no ***currentUser***, then the screen should redirect to the ***Signin*** screen. If there's a ***currentUser*** then the ***Profile*** screen should populate a form with the user's information. If the current user clicks a ***Sign Out*** button, then the current user should be nulled and navigate to the ***Signin*** screen. Refactor the ***Profile*** screen as shown below.

src/Kanbas/Account/Profile.tsx

```
import { Link, useNavigate } from "react-router-dom";
import { useState, useEffect } from "react";
import { useSelector, useDispatch } from "react-redux";
import { setCurrentUser } from "./reducer";
export default function Profile() {
  const [profile, setProfile] = useState<any>({});
  const dispatch = useDispatch();
  const navigate = useNavigate();
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  const fetchProfile = () => {
    if (!currentUser) return navigate("/Kanbas/Account/Signin");
    setProfile(currentUser);
  };
  const signout = () => {
    dispatch(setCurrentUser(null));
    navigate("/Kanbas/Account/Signin");
  };
  useEffect(() => { fetchProfile(); }, []);
  return (
    <div className="wd-profile-screen">
      <h3>Profile</h3>
      {profile && (
        <div>
          <input defaultValue={profile.username} id="wd-username" className="form-control mb-2"
                 onChange={(e) => setProfile({ ...profile, username:  e.target.value })}/>
          <input defaultValue={profile.password} id="wd-password" className="form-control mb-2"
                 onChange={(e) => setProfile({ ...profile, password:  e.target.value })}/>
          <input defaultValue={profile.firstName} id="wd-firstname" className="form-control mb-2"
                 onChange={(e) => setProfile({ ...profile, firstName: e.target.value })}/>
          <input defaultValue={profile.lastName} id="wd-lastname" className="form-control mb-2"
                 onChange={(e) => setProfile({ ...profile, lastName:  e.target.value })}/>
          <input defaultValue={profile.dob} id="wd-dob" className="form-control mb-2"
                 onChange={(e) => setProfile({ ...profile, dob: e.target.value })} type="date"/>
          <input defaultValue={profile.email} id="wd-email" className="form-control mb-2"
                 onChange={ (e) => setProfile({ ...profile, email: e.target.value })}/>
```

```
          <select onChange={(e) => setProfile({ ...profile, role:  e.target.value })}
                 className="form-control mb-2" id="wd-role">
            <option value="USER">User</option>           <option value="ADMIN">Admin</option>
            <option value="FACULTY">Faculty</option>     <option value="STUDENT">Student</option>
          </select>
          <button onClick={signout} className="btn btn-danger w-100 mb-2" id="wd-signout-btn">
            Sign out
          </button>
        </div>
      )}
</div>);}
```

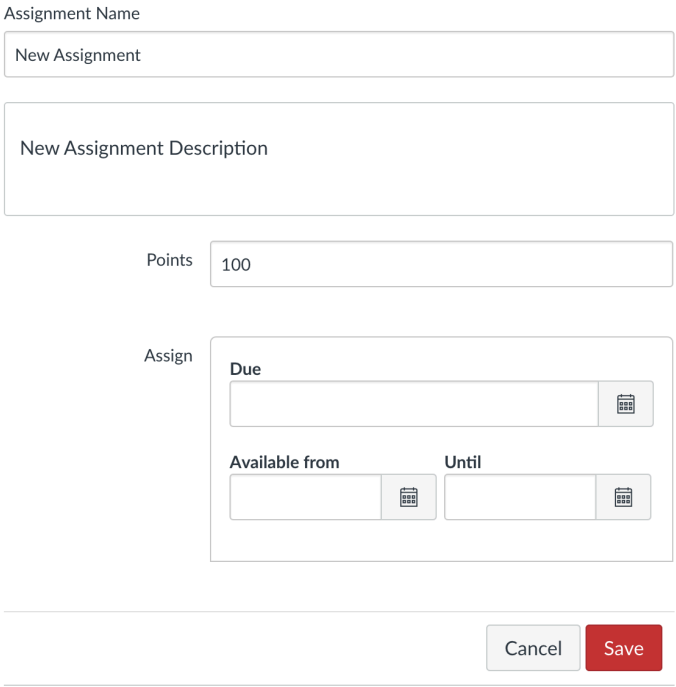
3.5 Assignments (On Your Own)



After completing the ***Dashboard***, ***Courses***, and ***Modules***, refactor the ***Assignments*** and ***AssignmentEditor*** screens to create, update, and remove assignments as described in this section.

3.5.1 Assignments Reducer

Following ***Modules/reducer.ts*** as an example, create an ***Assignments/reducer.ts*** in ***src/Kanbas/Courses/Assignments/*** initialized with ***db.assignments***. Implement reducer functions such as ***addAssignment***, ***deleteAssignment***, ***updateAssignment***, and any other functions as needed. Add the new reducer to the store in ***Kanbas/store/index.ts*** to add the assignments to the ***Kanbas*** application state.



3.5.2 Creating an Assignment

Refactor your ***Assignments*** screen as follows

- Clicking the ***+ Assignment*** button navigates to the ***AssignmentEditor*** screen
- The ***AssignmentEditor*** should allow editing at least the following fields: ***name***, ***description***, ***points***, ***due date***, ***available from date***, and ***available until date***.
- Clicking ***Save*** creates the new assignment and adds it to the ***assignments*** array state variable, navigates to the ***Assignments*** screen, which must now contain the newly created assignment.
- Clicking ***Cancel*** does not create the new assignment, and navigates back to the ***Assignments*** screen, without the new assignment.

3.5.3 Editing an Assignment

Refactor the ***AssignmentsEditor*** component as follows

- Clicking on an assignment in the ***Assignments*** screen navigates to the ***AssignmentsEditor*** screen, displaying the assignment's ***name***, ***description***, ***points***, ***due date***, ***available from date***, and ***available until date*** of the corresponding assignment.
- The ***AssignmentsEditor*** screen should allow editing the same fields listed earlier for corresponding assignment.
- Clicking ***Save*** updates the assignment's fields and navigates back to the ***Assignments*** screen with the updated assignment values.

- Clicking ***Cancel*** does not update the assignment, and navigates back to the ***Assignments*** screen which shows the assignments unchanged.

3.5.4 Deleting an Assignment

Refactor the ***Assignments*** component as follows

- Using the example of deleting modules, add a ***Delete*** button or ***trash*** icon to the right of each assignment.
- Clicking ***Delete*** on an assignment pops up a dialog asking if you are sure you want to remove the assignment.
- Clicking ***Yes*** or ***Ok***, dismisses the dialog, removes the assignment, and updates the ***Assignments*** screen without the deleted assignment.
- Clicking ***No*** or ***Cancel***, dismisses the dialog without removing the assignment

3.6 Enrollments (On Your Own)

Currently the ***Dashboard*** screen allows ***Faculty*** to ***Add***, ***Delete***, ***Edit***, and ***Update*** courses, as well as navigate to the course's content. Other types of the users can only view the list of the courses they are enrolled in. Refactor the ***Dashboard*** screen so that if the current user's role is ***Student***, they have a blue ***Enrollments*** button at the top right of the screen. Clicking the ***Enrollments*** button displays all the the courses. Clicking it again only shows the courses a student is enrolled in. Courses that the student is enrolled in should provide a red ***Unenroll*** button and courses that the student is not enrolled in should provide a green ***Enroll*** button. When a student click's the ***Unenroll*** or ***Enroll*** button the enrollment status must actually change and the buttons should toggle to reflect the new state. If a student signs out, and then signs in again, the enrollment choices should still persist. If a user refreshes or reloads the page, the new enrollments are lost. Protect the route to a course so that only students enrolled in that course can navigate to the course, and stay in the ***Dashboard*** screen otherwise. All enrollment related buttons should only be visible to students. Create new or modify existing reducers and store as needed.

4 Deliverables

1. In the same React.js application created in earlier assignments, ***kanbas-react-web-app***, complete all the exercises described in this document.
2. In a branch called ***a4***, add, commit and push the source code of the React.js application ***kanbas-react-web-app*** to the same remote source repository in ***GitHub.com*** created in an earlier assignment. Here's an example of how to add, commit and push your code

```
$ git checkout -b a4
$ git add .
$ git commit -am "a4 Redux"
$ git push
```

3. Deploy the ***a4*** branch to the same ***Netlify*** project created in an earlier assignment. Configure Netlify to deploy all branches to separate URLs. From your Netlify's dashboard go to ***Site settings > Build & deploy > Branches > Branch deploys*** and select ***All***. Now each time you commit to a branch, the application will be available at a URL that contains the name of the branch
4. Make sure ***Labs/index.tsx*** contains a ***TOC.tsx*** that references each of the labs and Kanbas. Add a link to your repository in GitHub.
5. In ***Labs/index.tsx***, add your full name: first name first, and last name second. Use the same name as in Canvas.
6. As a deliverable in ***Canvas***, submit the URL to the ***a4*** branch deployment of your React.js application running on Netlify.