Managing State in React.js Application

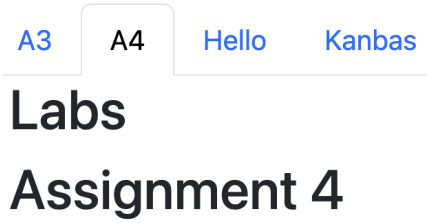
1 Introduction

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 Assignment4 Component



To get started, create an ***Assignment4*** component that will host all the exercises in this assignment. Then import the component into the ***Labs*** component created in an earlier assignment. If not done already, add routes in ***Labs*** so that each assignment will appear in its own screen when you navigate to ***Labs*** and then to ***/a4***. Make the ***Assignment3*** component the default element that renders when navigating to ***http://localhost:3000/#/Labs*** path and map ***Assignment4*** to the ***/a4*** path. You might need to change the lab component route in ***App.tsx*** so that all routes after ***/Labs/**** are handled by the routes declared in the ***Labs*** component, e.g., `<Route path="/Labs/*" element={<Labs/>}/>`. You might also want to make ***Assignment3*** the default component by changing the ***to*** attribute in the ***Navigate*** component in ***App.tsx***, e.g., `<Route path="/" element={ <Navigate to="a3"/>}/>`. Use the code snippets below as a guide.

src/Labs/a4/index.tsx

```
import React from "react";

const Assignment4 = () => {
  return(
    <>
      <h1>Assignment 4</h1>
    </>
  );
};
export default Assignment4;
```

src/Nav.tsx

```
import { Link } from "react-router-dom";

function Nav() {
 return (
   <nav className="nav nav-pills mb-2">
     <Link className="nav-link" to="/Labs/a3">
       A3</Link>
     <Link className="nav-link" to="/Labs/a4">
       A4</Link>
     <Link className="nav-link" to="/hello">
       Hello</Link>
     <Link className="nav-link" to="/Kanbas">
```

src/Labs/index.tsx

```
import Nav from "../Nav";
import Assignment3 from "./a3";
import Assignment4 from "./a4";
import {Routes, Route, Navigate}
  from "react-router";
function Labs() {
 return (
   <div>
     <Nav/>
     <Routes>
       <Route path="/"
        element={<Navigate
```

```
      Kanbas</Link>
    </nav>
  );
}

export default Nav;
```

```
                to="a3"/>}/>
        <Route path="a3"
         element={<Assignment3/>}/>
        <Route path="a4"
         element={<Assignment4/>}/>
      </Routes>
    </div>
  );
}
export default Labs;
```

2.2 Handling User Events

2.2.1 Handling Click Events

HTML elements can handle mouse clicks using the ***onClick*** to declare a function to handle the event. The example below calls function ***hello*** when you click the ***Click Hello*** button. Add the component to ***Assignment4*** and confirm it behaves as expected.

src/Labs/a4/ClickEvent.tsx

```
function ClickEvent() {
  const hello = () => {
    alert("Hello World!");
  };
  const lifeIs = (good: string) => {
    alert(`Life is ${good}`);
  };
  return (
    <div>
      <h2>Click Event</h2>
      <button onClick={hello}>
        Click Hello</button>
      <button onClick={() => lifeIs("Good!")}>
        Click Good</button>
      <button
        onClick={() => {
          hello();
          lifeIs("Great!");
        }}
      >
        Click Hello 3
      </button>
    </div>
  );
}
export default ClickEvent;
```

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

// 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()
```

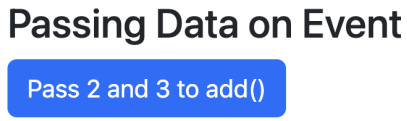
2.2.2 Passing Data when Handling Events

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 ***Assignment4*** and confirm it works as expected.

src/Labs/a4/PassingDataOnEvent.tsx

```
const add = (a: number, b: number) => {
  alert(`${a} + ${b} = ${a + b}`);
};
function PassingDataOnEvent() {
  return (
```

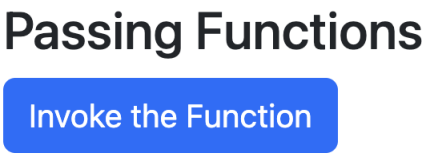
```
// function expects a and b
```



```
    <div>
      <h2>Passing Data on Event</h2>
      <button onClick={() => add(2, 3)}
              // onClick={add(2, 3)}
              className="btn btn-primary">
        Pass 2 and 3 to add()
      </button>
    </div>
  );
}
export default PassingDataOnEvent;
```

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

2.2.3 Passing Functions as Attributes



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/a4/PassingFunctions.tsx

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

```
// function passed in as a parameter


// invoking function
```

Include the component in ***Assignment4***, declare a ***sayHello*** callback function, pass it to the ***PassingFunctions*** component, and confirm it works as expected.

src/Labs/a4/index.tsx

```
import PassingFunctions from "./PassingFunctions";
function Assignment4() {
  function sayHello() {
    alert("Hello");
  }
  return (
    <div>
      <h1>Assignment 4</h1>
      <PassingFunctions theFunction={sayHello} />
      ...
    </div>
  );
}
export default Assignment4;
```

```
// 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 ***Assignment4***, click the button and confirm the event object is rendered on the screen.

src/Labs/a4/EventObject.tsx

```
import React, { useState } from "react";
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 id="event-button"
        onClick={(e) => handleClick(e)}
        className="btn btn-primary">
        Display Event Object
      </button>
      <pre>{JSON.stringify(event, null, 2)}</pre>
    </div>
  );
}
export default EventObject;
```

```
// 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/a4/Counter.tsx

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

```
// 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
```

Counter: 7

Up Down

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 ***Assignment4*** and confirm it works as expected. Do the same with the rest of the exercises that follow.

src/Labs/a4/Counter.tsx

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

```
// 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
```

Counter: 7

Up Down

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.

Boolean State Variables

Done

Done

Yay! you are done

src/Labs/a4/BooleanStateVariables.tsx

```
import React, { useState } from "react";
function BooleanStateVariables() {
  const [done, setDone] = useState(true);
  return (
    <div>
      <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>}
    </div>
  );
}
export default BooleanStateVariables;
```

```
// 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.

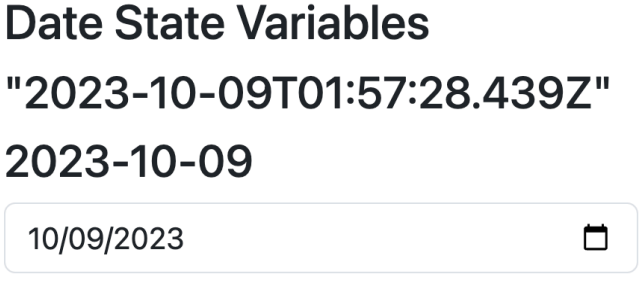
src/Labs/a4/StringStateVariables.tsx

```
import React, { useState } from "react";
function StringStateVariables() {
  const [firstName, setFirstName] = useState("John");
  return (
    <div>
      <h2>String State Variables</h2>
      <p>{firstName}</p>
      <input
        className="form-control"
        value={firstName}
        onChange={(e) => setFirstName(e.target.value)}/>
    </div>
  );
}
export default StringStateVariables;
```

```
// import useState

// declare and
// initialize
// state variable

// render string
// 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.



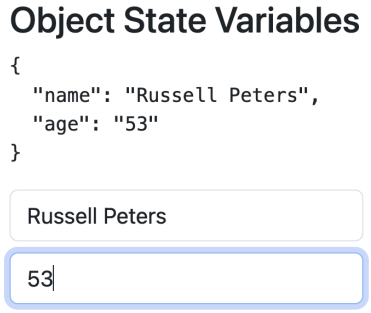
src/Labs/a4/DateStateVariable.tsx

```tsx
import React, { useState } from "react";
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>
      <h2>Date State Variables</h2>
      <h3>{JSON.stringify(startDate)}</h3>
      <h3>{dateObjectToHtmlDateString(startDate)}</h3>
      <input
        className="form-control"
        type="date"
        value={dateObjectToHtmlDateString(startDate)}
        onChange={(e) => setStartDate(new Date(e.target.value))}
      />
    </div>
  );
}
export default DateStateVariable;
```

```tsx
// 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/a4/ObjectStateVariable.tsx

```tsx
import React, { useState } from "react";
function ObjectStateVariable() {
  const [person, setPerson] = useState({ name: "Peter", age: 24 });
  return (
    <div>
      <h2>Object State Variables</h2>
      <pre>{JSON.stringify(person, null, 2)}</pre>
      <input
        value={person.name}
        onChange={(e) => setPerson({ ...person, name: e.target.value })}
      />
      <input
        value={person.age}
        onChange={(e) => setPerson({ ...person,
                                     age: parseInt(e.target.value) })}
      />
    </div>
  );
}
export default ObjectStateVariable;
```

```tsx
// import useState

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

// 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

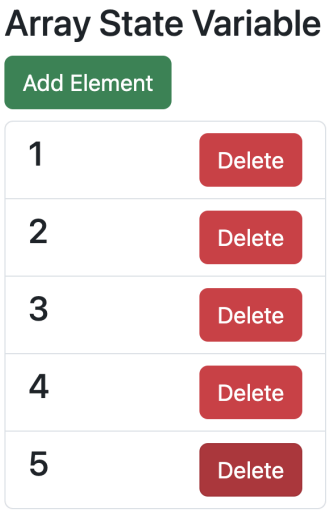
The ***ArrayStateVariable*** component below demonstrates how to work with ***array*** state variables. An array of integers if declared as a state variable and function ***addElement*** and ***deleteElement*** are used to add and remove elements to and

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.

src/Labs/a4/ArrayStateVariable.tsx

```
import React, { useState } from "react";
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>
      <h2>Array State Variable</h2>
      <button onClick={addElement}>Add Element</button>
      <ul>
        {array.map((item, index) => (
          <li key={index}>
            {item}
            <button onClick={() => deleteElement(index)}>
              Delete</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
export default ArrayStateVariable;
```

```
// 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/a4/ParentStateComponent.tsx

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

```
//
```

The ***ChildStateComponent*** can use references to ***counter*** and ***setCounter*** to render the state variable and manipulate it through the mutator function. Import ***ParentStateComponent*** into ***Assignment4*** and confirm it works as expected.

src/Labs/a4/ChildStateComponent.tsx

```
function ChildStateComponent({ counter, setCounter }:
{ counter: number;
  setCounter: (counter: number) => void;}) {
  return (
    <div>
      <h3>Counter {counter}</h3>
      <button onClick={() => setCounter(counter + 1)}>
        Increment</button>
      <button onClick={() => setCounter(counter - 1)}>
        Decrement</button>
    </div>
  );
}
export default ChildStateComponent;
```

```
//
```

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/a4/ReduxExamples/index.tsx*** as shown below. Import the new redux examples component into the assignment 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/a4/ReduxExamples/index.tsx

```
import React from "react";

const ReduxExamples = () => {
  return(
    <div>
      <h2>Redux Examples</h2>
    </div>
  );
};

export default ReduxExamples;
```

src/Labs/a4/index.tsx

```
import React from "react";
import ReduxExamples from "./redux-examples";

const Assignment4 = () => {
 return(
   <>
     <h1>Assignment 4</h1>
     <ReduxExamples/>
     ...
   </>
 );
};
export default Assignment4;
```

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/a4/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.tsx

```
import { configureStore } from "@reduxjs/toolkit";
import helloReducer from "../a4/ReduxExamples/HelloRedux/helloReducer";
export interface LabState {
  helloReducer: {
    message: string;
  };
}
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";
function Labs() {
  return (
    <Provider store={store}>
      <div className="container-fluid">
        <h1>Labs</h1>
        ...
      </div>
    </Provider>
  );
}
export default Labs;
```

```
//
```

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 works as expected.

src/Labs/a4/ReduxExamples/HelloRedux/index.tsx

```
import { useSelector, useDispatch } from "react-redux";
import { LabState } from "../../../store";
function HelloRedux() {
  const { message } = useSelector((state: LabState) => state.helloReducer);
  return (
    <div>
      <h1>Hello Redux</h1>
      <h2>{message}</h2>
    </div>
  );
}
export default HelloRedux;
```

```
//
```

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/a4/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 "../a4/ReduxExamples/HelloRedux/helloReducer";
import counterReducer from "../a4/ReduxExamples/CounterRedux/counterReducer";
export interface LabState {
  helloReducer: { message: string; };
  counterReducer: {
    count: number;
  };
}
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/a4/ReduxExamples/CounterRedux/index.tsx

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

```
//
```

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/a4/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 "../a4/ReduxExamples/HelloRedux/helloReducer";
import counterReducer from "../a4/ReduxExamples/CounterRedux/counterReducer";
import addReducer from "../a4/ReduxExamples/AddRedux/addReducer";
export interface LabState {
  helloReducer: { message: string; };
  counterReducer: { count: number; };
  addReducer: {
    sum: number;
  };
}
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.

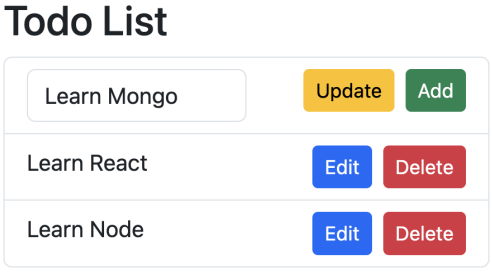
src/Labs/a4/ReduxExamples/AddRedux/index.tsx

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

```
// 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

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.



src/Labs/a4/ReduxExamples/todos/TodoList.tsx

```
import React, { useState } from "react";
function TodoList() {
  const [todos, setTodos] = useState([
    { id: "1", title: "Learn React" },
    { id: "2", title: "Learn Node"  }]);
```

```
// import useState

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

```
  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: ""});
  };
  return (
    <div>
      <h2>Todo List</h2>
      <ul className="list-group">
        <li className="list-group-item">
          <button onClick={() => addTodo(todo)}>Add</button>
          <button onClick={() => updateTodo(todo)}>
            Update </button>
          <input
            value={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)}>
              Delete </button>
            <button onClick={() => setTodo(todo)}>
              Edit </button>
            {todo.title}
          </li>
        ))}
      </ul>
    </div>
  );
}
export default TodoList;
```

```
// 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


// 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/a4/ReduxExamples/todos/TodoItem.tsx

```
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)}> Delete </button>
      <button onClick={() => setTodo(todo)}> Edit </button>
      {todo.title}
    </li>
  );
}
export default TodoItem;
```

```
// 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/a4/ReduxExamples/todos/TodoForm.tsx

```
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">
      <button onClick={() => addTodo(todo)}> Add </button>
      <button onClick={() => updateTodo(todo)}> Update </button>
      <input
        value={todo.title}
        onChange={ (e) => setTodo({ ...todo, title: e.target.value }) }
      />
    </li>
  );
}
export default TodoForm;
```

```
// 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



// 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 ***Assignment4*** and confirm it works as expected.

src/Labs/a4/ReduxExamples/todos/TodoList.tsx

```
import React, { useState } from "react";
import TodoForm from "./TodoForm";
import TodoItem from "./TodoItem";
function TodoList() {
  ...
  return (
    <div>
      <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>
    </div>
  );
}
export default TodoList;
```

```
// 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 ***Assignment3*** component or ***Kanbas***? We would have to move the todos state variable and mutator functions to a component that is parent to both the ***Assignment3*** component and the ***TodoList*** component, e.g., ***Labs***.

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/a4/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 = [
          ...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;
```

```
// 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
// 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 "../a4/ReduxExamples/HelloRedux/helloReducer";
import counterReducer from "../a4/ReduxExamples/CounterRedux/counterReducer";
import addReducer from "../a4/ReduxExamples/AddRedux/addReducer";
import todosReducer from "../a4/ReduxExamples/todos/todosReducer";
export type TodoType = {
  id: string;
  title: string;
};
export interface LabState {
  ...
  todosReducer: {
    todos: TodoType[];
    todo: TodoType;
  };
}
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 ***todo*** parameter.

src/Labs/a4/ReduxExamples/todos/TodoForm.tsx

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

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

```
// 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/a4/ReduxExamples/todos/TodoItem.tsx

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

```
// 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/a4/ReduxExamples/todos/TodoList.tsx

```
import React from "react";
import TodoForm from "./TodoForm";
import TodoItem from "./TodoItem";
import { useSelector } from "react-redux";
import { LabState, TodoType } from "../../../store";
function TodoList() {
  const { todos } = useSelector((state: LabState) => state.todosReducer);
  return (
```

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

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

```
    <div>
      <h2>Todo List</h2>
      <ul className="list-group">
        <TodoForm />
        {todos.map((todo: TodoType) => (
          <TodoItem todo={todo} />
        ))}
      </ul>
    </div>
  );
}
export default TodoList;
```

```
// 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 ***Assignment3*** component as shown below and confirm the todos display in ***Assignment3***.

src/Labs/a3/index.tsx

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

function Assignment3() {
  const { todos } = useSelector((state: LabState) => state.todosReducer);
  return (
    <div>
      <h2>Assignment 3</h2>
      <ul className="list-group">
        {todos.map((todo) => (
          <li className="list-group-item" key={todo.id}>
            {todo.title}
          </li>
        ))}
      </ul>
      ...
    </div>
  );
}
export default Assignment3;
```

```
//
```

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 Dashboard

The current ***Dashboard*** implementation renders a static array of courses. Let's refactor the ***Dashboard*** so we can create new courses, update existing course titles, and remove 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. The screenshot here on the right, gives an idea of the implementation suggested in the following exercises. Use your existing HTML and CSS to render the courses with Bootstrap as you did for previous assignments. Add a form similar to the one suggested here, as well as ***Edit*** and ***Delete*** buttons to each of the courses. Feel free to style the new buttons and form as you like.

src/Kanbas/Dashboard/index.tsx

```
import React, { useState } from "react";                          // add useState hook
import { Link } from "react-router-dom";
import db from "../Database";
function Dashboard() {
  const [courses, setCourses] = useState(db.courses);             // create courses state
  return (                                                        // variable and initialize
    <div className="p-4">                                         // with database's courses
      <h1>Dashboard</h1> <hr />
      <h2>Published Courses ({courses.length})</h2> <hr />
      <div className="row">
        <div className="row row-cols-1 row-cols-md-5 g-4">
          {courses.map((course) => (                              // use courses state variable instead of
            <div key={course._id} className="col" style={{ width: "300px" }}>  // the database courses variable
              <div className="card">                              // reuse the same HTML you used in
                 ...  {course.name}   ...                         // previous assignments
              </div>
            </div>
          ))}
        </div>
      </div>
    </div>
  );
}
export default Dashboard;
```

3.1.1 Creating New Courses

To create new courses, implement ***addNewCourse*** function as shown below and 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. Confirm you can add new courses.

src/Kanbas/Dashboard/index.tsx

```
...
function Dashboard() {
  const [courses, setCourses] = useState(db.courses);
  const course = {                                                // create a course object with default values
    _id: "0", name: "New Course", number: "New Number",
    startDate: "2023-09-10", endDate: "2023-12-15",
    image: "/images/reactjs.jpg"
  };
  const addNewCourse = () => {                                    // create addNewCourse event handler that sets
    const newCourse = { ...course,                                // courses as copy of current courses state array
                        _id: new Date().getTime().toString() };   // add course at the end of the array
    setCourses([...courses, { ...course, ...newCourse }]);        // overriding _id to current time stamp
  };
  return (
    <div>
      <h1>Dashboard</h1>
      <button onClick={addNewCourse} >                            // add button to invoke
        Add                                                       // addNewCourse. Note no argument syntax
      </button>
      ...
    </div>
  );
}
```

Use the ***course*** constant as the initial state of a new state variable of the same name as shown below. Add a form to edit the ***course*** state variable's ***name***, ***number***, ***startDate***, and ***endDate***. Confirm form shows values of the ***course*** state variable.

src/Kanbas/Dashboard/index.tsx

```
...
function Dashboard() {
  const [courses, setCourses] = useState(db.courses);
  const [course, setCourse] = useState({
    _id: "0", name: "New Course", number: "New Number",
    startDate: "2023-09-10", endDate: "2023-12-15",
    image: "/images/reactjs.jpg"
  });
  const addNewCourse = () => { ... };
  return (
    <div>
      <h1>Dashboard</h1>
      <h5>Course</h5>
      <input value={course.name} className="form-control" />
      <input value={course.number} className="form-control" />
      <input value={course.startDate} className="form-control" type="date" />
      <input value={course.endDate} className="form-control" type="date" />
      <button onClick={addNewCourse} >
        Add
      </button>
      ...
    </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 as below. Use your implementation of ***Dashboard*** and use the code provided as an example. Confirm you can add edit and new courses.

src/Kanbas/Dashboard/index.tsx

```
...
function Dashboard() {
  const [courses, setCourses] = useState(db.courses);
  const [course, setCourse] = useState({ ... });
  const addNewCourse = () => { ... };
  return (
    <div>
      <h1>Dashboard</h1>
      <h5>Course</h5>
      <input value={course.name} className="form-control"
             onChange={(e) => setCourse({ ...course, name: e.target.value }) } />
      <input value={course.number} className="form-control"
             onChange={(e) => setCourse({ ...course, number: e.target.value }) } />
      <input value={course.startDate} className="form-control" type="date"
             onChange={(e) => setCourse({ ...course, startDate: e.target.value }) }/>
      <input value={course.endDate} className="form-control" type="date"
             onChange={(e) => setCourse({ ...course, endDate: e.target.value }) } />
      <button onClick={addNewCourse} >
        Add
      </button>
      ...
    </div>
  );
}
```

```
// add onChange event
// handlers to each input
// element to update
// course state with
// event's target value
```

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/index.tsx

```
...
function Dashboard() {
  const [courses, setCourses] = useState(db.courses);
  const [course, setCourse] = useState({ ... });
  const addNewCourse = () => { ... };
  const deleteCourse = (courseId: string) => {
    setCourses(courses.filter((course) => course._id !== courseId));
  };
  return (
    <div>
      <h1>Dashboard</h1>
      ...
      <div className="row">
        <div className="row row-cols-1 row-cols-md-5 g-4">
          {courses.map((course) => (
            ...
            <Link className="card-title"
                  to={`/Kanbas/Courses/${course._id}`}>
                    {course.name}
              <button onClick={(event) => {
                        event.preventDefault();
                        deleteCourse(course._id);
                      }}>
                      Delete
              </button>
            </Link>
            ...
          ))}
        </div>
      </div>
    </div>
  );
}
export default Dashboard;
```

```
// 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*** of a course, copies the course into the form.

src/Kanbas/Dashboard/index.tsx

```
...
        {courses.map((course) => (
          <Link key={course._id}>
                to={`/Kanbas/Courses/${course._id}`}
                className="list-group-item">
            <button onClick={(event) => {
                event.preventDefault();
                setCourse(course);
              }}>
              Edit
            </button>
            <button
              onClick={(event) => {
                event.preventDefault();
                deleteCourse(course._id);
              }}>
              Delete
            </button>
            {course.name}
          </Link>
        ))}
```

```
// add 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 a ***Update*** button to the form so that the selected course is 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.

src/Kanbas/Dashboard/index.tsx

```
...
function Dashboard() {
  const [courses, setCourses] = useState(db.courses);
  const [course, setCourse] = useState({ ... });
  const updateCourse = () => {
    setCourses(
      courses.map((c) => {
        if (c._id === course._id) {
          return course;
        } else {
          return c;
        }
      })
    );
  };
  return (
    <div>
      <h1>Dashboard</h1>
      <h5>Course</h5>
      <input value={course.name} className="form-control".../>
      <input value={course.number} className="form-control".../>
      <input value={course.startDate} className="form-control".../>
      <input value={course.endDate} className="form-control".../>
      <button onClick={addNewCourse} >
        Add
      </button>
      <button onClick={updateCourse} >
        Update
      </button>
      ...
    </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 anywhere, or move the ***courses*** state variable to a component that contains 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/index.tsx

```
...
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>
      <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 db from "./Database";                                    // import the database
import { useState } from "react";                               // import the useState hook

function Kanbas() {
  const [courses, setCourses] = useState<any[]>(db.courses);    // move the state variables here
  const [course, setCourse] = useState({                        // from the Dashboard
    _id: "1234", name: "New Course", number: "New Number",
    startDate: "2023-09-10", endDate: "2023-12-15",
  });
  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 className="d-flex">
      <KanbasNavigation />
      <div>
        <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/:courseId/*" 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
  );
}
export default Kanbas;
```

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***.

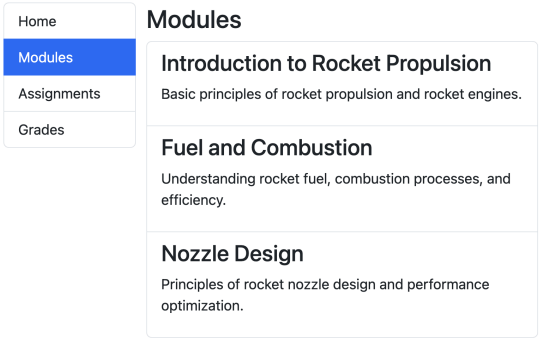
src/Kanbas/Courses/index.tsx

```
...
function Courses({ courses }: { courses: any[]; }) {          // accept courses from Kanbas
```

```
  const { courseId } = useParams();
  const course = courses.find((course) => course._id === courseId);
  return (...);
}
export default Courses;
```

```
// find the course by its ID
```

3.3 Modules



Now let's do the same with ***Modules***. We'll first refactor the ***ModuleList*** component using component 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 ***ModuleList***. 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 ***ModuleList*** implementation by converting the ***modules*** array into a state variable as shown below. Confirm ***ModuleList*** renders as expected. Styling shown here is for illustration purposes. Use your HTML and CSS from previous assignments to style the modules.

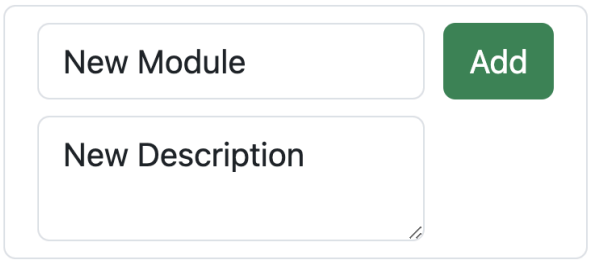
src/Kanbas/Courses/Modules/List.tsx

```
import React, { useState } from "react";
import { useParams } from "react-router-dom";
import { modules } from "../../Database";

function ModuleList() {
  const { courseId } = useParams();
  const [moduleList, setModuleList] = useState<any[]>(modules);
  return (
    <>
    <ul className="list-group wd-modules">
      {moduleList
        .filter((module) => module.course === courseId)
        .map((module, index) => (
          <li key={index} className="list-group-item">
            ...
            {module.name}
            <p>{module.description}</p>
            <p>{module._id}</p>
            ...
          </li>))}
    </ul>
    </>
  );
}
export default ModuleList;
```

```
// import useState to create
// state variables
```

```
// create modules state variables
// initialized from db
```

3.3.1 Creating a Module



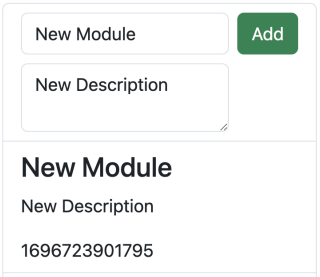
Add a new ***module*** state variable and corresponding form to edit and create new module names and titles as shown below. Refactor your ***ModuleList*** component as suggested below and confirm the form renders the ***module*** state variable as expected. Reuse the HTML and CSS from previous assignments to style the modules. You can use the styling suggested here for the form and buttons, but feel free to come up with your own unique styling.

src/Kanbas/Courses/Modules/List.tsx

```
...
function ModuleList() {
  const { courseId } = useParams();
  const [moduleList, setModuleList] = useState<any[]>(modules);
  const [module, setModule] = useState({
    name: "New Module",
    description: "New Description",
    course: courseId,
  });
  return (
    <>
    <ul className="list-group wd-modules">
      <li className="list-group-item">
        <button>Add</button>
        <input value={module.name}
          onChange={(e) => setModule({
            ...module, name: e.target.value })}
        />
        <textarea value={module.description}
          onChange={(e) => setModule({
            ...module, description: e.target.value })}
        />
      </li>
      {moduleList
        .filter((module) => module.course === courseId)
        .map((module, index) => (
          <li key={index} className="list-group-item">
            ...
          </li>))}
    </ul>
    </>
  );
}
export default ModuleList;
```

```
// declare module state variable initialized with
// default values for name, description, and course
// used to edit new and existing modules

// add a form to edit the module
// Add button to add the new module
// input field to edit module's name. default
// value from module.name. update module.name for
// every key stroke

// textarea to edit module's description. default
// value from module.description. update description
// for every key stroke
```

Implement a new ***addModule*** function that appends a new module at the end of the ***modules*** state variable. Confirm you can add new modules. Reuse the HTML and CSS from previous assignments to style the modules. You can use the styling suggested here for the form and buttons, but feel free to come up with your own unique styling.



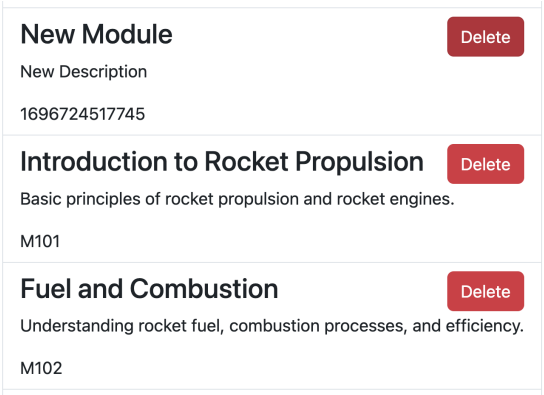
src/Kanbas/Courses/Modules/List.tsx

```
function ModuleList() {
  ...
  const [module, setModule] = useState({
    _id: "0", name: "New Module",
    description: "New Description",
    course: courseId || "",
  });
  const addModule = (module: any) => {
    const newModule = { ...module,
      _id: new Date().getTime().toString() };
    const newModuleList = [newModule, ...moduleList];
    setModuleList(newModuleList);
  };
  return (
    <>
    <ul className="list-group wd-modules">
      <li className="list-group-item">
        <button onClick={() => { addModule(module) }}>
           Add
        </button>
        ...
      </li>
      ...
    </ul>
  );
}
export default ModuleList;
```

```
// addModule appends new module at beginning of
// modules, overriding _id with a timestamp

// Add button calls addModule with module being
// edited in the form to be added to the modules
```

3.3.2 Deleting a Module

Add ***Delete*** buttons to each module that invokes a new ***deleteModule*** function passing the ID of the module we want to remove. The new function should filter out the module and create a new array without the module we are deleting. Refactor ***ModuleList*** as suggested below and confirm you can remove modules. Styling shown here is for illustration purposes. Use your HTML and CSS from previous assignments to style the modules.



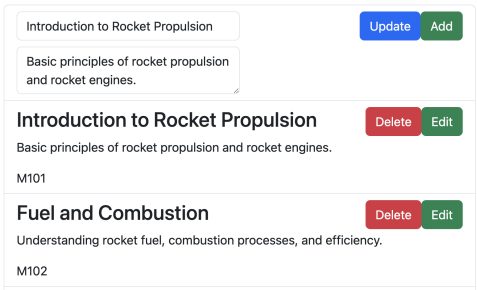
src/Kanbas/Courses/Modules/List.tsx

```
function ModuleList() {
  ...
  const [module, setModule] = useState({ ... });
  const addModule = () => { ... };
  const deleteModule = (moduleId: string) => {
    const newModuleList = moduleList.filter(
      (module) => module._id !== moduleId );
    setModuleList(newModuleList);
  };
  return (
    <>
    <ul className="list-group wd-modules">
      ...
      {moduleList
        .filter((module) => module.course === courseId)
        .map((module, index) => (
          <li key={index} className="list-group-item">
            <button
              onClick={() => deleteModule(module._id)}>
              Delete
            </button>
            ...
            {module.name}
            <p>{module.description}</p>
            ...
      </li>
      ...
    </ul>
  );
}
export default ModuleList;
```

```
// deleteModule filters out the module whose ID is
// equal to the parameter moduleId

// delete button calls deleteModule with module's ID
// to be removed
```

3.3.3 Editing a Module

Add an ***Edit*** button to each of the modules that copies the corresponding module to the form as shown below. Also add a new ***Update*** button to the form which computes a new ***modules*** array that replaces the module being edited with the updates in the form. Confirm you can edit modules. Styling shown here is for illustration purposes. Use your HTML and CSS from previous assignments to style the modules.



src/Kanbas/Courses/Modules/ModuleList.tsx

```
function ModuleList() {
  const [module, setModule] = useState({ ... });
  ...
  const updateModule = () => {
    const newModuleList = moduleList.map((m) => {
      if (m._id === module._id) {
        return module;
      } else {
        return m;
      }
    });
    setModuleList(newModuleList);
```

```
  };
  return (
    <ul className="list-group">
      <li className="list-group-item">
        <button onClick={addModule}>Add</button>
        <button onClick={updateModule}>
                Update
        </button>
        ...
      </li>
      {moduleList
        .filter((module) => module.course === courseId)
        .map((module, index) => (
          <li key={index} className="list-group-item">
            <button
              onClick={(event) => { setModule(module); }}>
              Edit
            </button>
            <button
              onClick={() => deleteModule(module._id)}>
              Delete
            </button>
            {module.name}
            ...
          </li>))}
    </ul>
  );
}
export default ModuleList;
```

```
// updateModule rebuilds modules by replacing the module
// whose ID matches the current module being edited

// update button calls updateModule

// edit button copies this module to current module
// so it can be edited
```

3.3.4 Module Reducer

The ***ModuleList*** seems to be working as expected being able to 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 ***ModuleList*** 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 ***moduleReducer.tsx*** shown below containing the ***modules*** and ***module*** state variables as well as the ***addModule***, ***deleteModule***, ***updateModule***, and ***setModule*** 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,
  module: { name: "New Module 123", description: "New Description" },
};

const modulesSlice = createSlice({
  name: "modules",
  initialState,
  reducers: {
    addModule: (state, action) => {
      state.modules = [
        { ...action.payload, _id: new Date().getTime().toString() },
          ...state.modules,
      ];
    },
    deleteModule: (state, action) => {
      state.modules = state.modules.filter(
        (module) => module._id !== action.payload
      );
    },
    updateModule: (state, action) => {
      state.modules = state.modules.map((module) => {
```

```
// import createSlice
// import modules from database

// create reducer's initial state with
// default modules copied from database
// default module

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

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

```
        if (module._id === action.payload._id) {
          return action.payload;
        } else {
          return module;
        }
      });
    },
    setModule: (state, action) => {
      state.module = action.payload;
    },
  },
});

export const { addModule, deleteModule,
  updateModule, setModule } = modulesSlice.actions;
export default modulesSlice.reducer;
```

```
// action.payload._id

// select the module to edit

// 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/index.ts

```
import { configureStore } from "@reduxjs/toolkit";
import modulesReducer from "../Courses/Modules/reducer";
export interface KanbasState {
  modulesReducer: {
    modules: any[];
    module: any;
  };
}
const store = configureStore({
  reducer: {
    modulesReducer
  }
});

export default store;
```

```
// configure a new store
// import reducer

// add reducer to 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";

function Kanbas() {
  ...
  return (
    <Provider store={store}>
      <div className="d-flex">
        <KanbasNavigation />
        <div>
          ...
        </div>
    </Provider>
  );
}
export default Kanbas;
```

```
// 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 ***ModuleList*** 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.

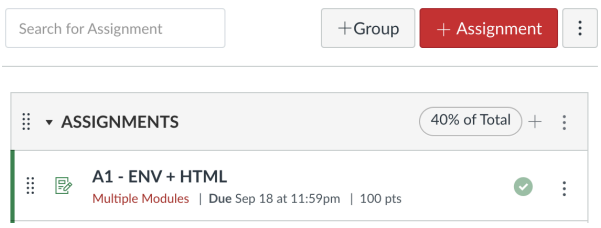
src/Kanbas/Courses/Modules/List.tsx

```
import React, { useState } from "react";
```

```
import { useParams } from "react-router-dom";
import { useSelector, useDispatch } from "react-redux";
import {
  addModule,
  deleteModule,
  updateModule,
  setModule,
} from "./reducer";
import { KanbasState } from "../../store";
function ModuleList() {
  const { courseId } = useParams();
  const moduleList = useSelector((state: KanbasState) =>
    state.modulesReducer.modules);
  const module = useSelector((state: KanbasState) =>
    state.modulesReducer.module);
  const dispatch = useDispatch();
  return (
    <ul className="list-group">
      <li className="list-group-item">
        <button
          onClick={() => dispatch(addModule({ ...module, course: courseId }))}>
          Add
        </button>
        <button
          onClick={() => dispatch(updateModule(module))}>
          Update
        </button>
        <input
          value={module.name}
          onChange={(e) =>
            dispatch(setModule({ ...module, name: e.target.value }))
          }/>
        <textarea
          value={module.description}
          onChange={(e) =>
            dispatch(setModule({ ...module, description: e.target.value }))
          }/>
      </li>
      {moduleList
        .filter((module) => module.course === courseId)
        .map((module, index) => (
          <li key={index} className="list-group-item">
            <button
              onClick={() => dispatch(setModule(module))}>
              Edit
            </button>
            <button
              onClick={() => dispatch(deleteModule(module._id))}>
              Delete
            </button>
            <h3>{module.name}</h3>
            <p>{module.description}</p>
          </li>
        ))}
    </ul>
  );
}
export default ModuleList;
```

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

// retrieve current state variables
// modules and module from reducer

// get dispatch to call reducer
// functions

// wrap reducer functions with
// dispatch

// wrap reducer functions with
// dispatch

// wrap reducer functions with
// dispatch

// wrap reducer functions with
// dispatch

// wrap reducer functions with
// dispatch

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

3.4 Assignments (graduates only)



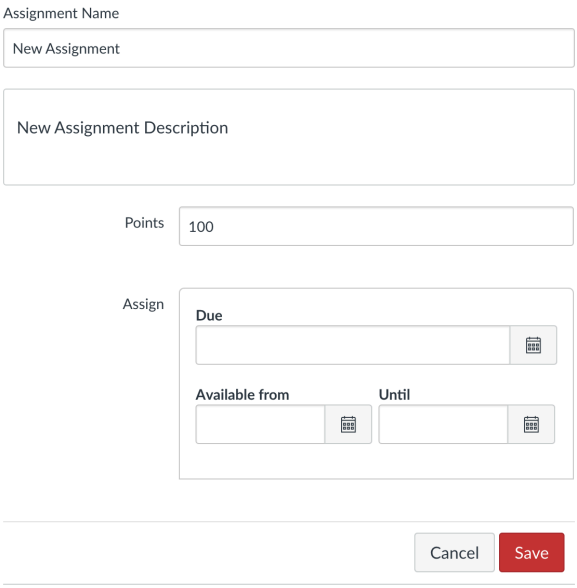
After completing the ***Dashboard***, ***Courses***, and ***ModuleList***, refactor the ***Assignments*** and ***AssignmentEditor*** screens to create, update, and remove assignments as described in this section.

3.4.1 Assignments Reducer

Following ***Modules/reducer.ts*** as an example, create an ***assignmentsReducer.ts*** in ***src/Kanbas/Courses/Assignments/*** initialized with ***db.assignments***. Implement reducer functions ***addAssignment***, ***deleteAssignment***, ***updateAssignment***, and

selectAssignment. Add the ***assignmentsReducer*** to the store in ***Kanbas/store/index.ts*** to add the assignments to the ***Kanbas*** application state

3.4.2 Creating an Assignment



Refactor your ***Assignments*** component as follows

- Clicking the ***+ Assignment*** button navigates to the ***AssignmentEditor*** screen
- The ***AssignmentEditor*** should allow editing the following fields: ***name***, ***description***, ***points***, ***dueDate***, ***availableFromDate***, ***availableUntilDate***.
- Clicking ***Save*** creates the new assignment and adds it to the ***assignments*** array state variable and displays in 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.

3.4.3 Editing an Assignment

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

- Clicking on an assignment in the ***Assignments*** screen navigates to the ***AssignmentsEditor*** screen, displaying the 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

3.4.4 Deleting an Assignment

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

- Add a ***Delete*** button 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

4 Deliverables

As a deliverable, make sure you complete the ***Labs*** and ***Kanbas*** sections of this assignment. All your work must be done in a branch called ***a4***. When done, add, commit and push the branch to GitHub. Deploy the new branch to Netlify and confirm it's available in a new URL based on the branch name. Submit the link to your GitHub repository and the new URL where the branch deployed to in Netlify. Here's an example on the steps:

*Create a branch called **a4***

```
git  checkout  -b  a4

# do all your work
```

Do all your work, e.g., ***Labs*** exercises, ***Kanbas***

Add, commit and push the new branch

```
git  add  .
git  commit  -am  "a4 State and Redux fa23"
git  push
```

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

1. The new URL where your ***a4*** branch deployed to on Netlify
2. The link to your new branch in GitHub