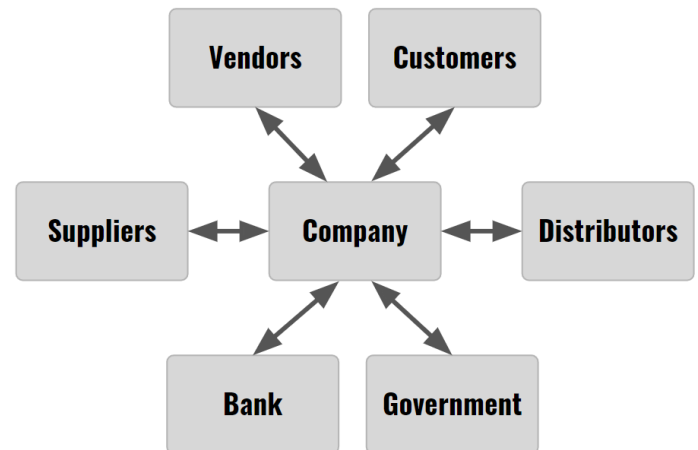


# Implementing HTTP Servers and RESTful APIs with Node.js

## 1 Introduction

During the 90s, the adoption of the World Wide Web grew exponentially. A variety of commercial ventures explored numerous use cases, revolutionizing interactions between companies, their customers, and other businesses. The figure on the right illustrates several integration points between businesses, often referred to as **business-to-business (B2B)** interactions. Interactions between businesses and customers are commonly known as **business-to-consumer (B2C)** interactions. Many companies have largely automated customer interactions by implementing online storefronts where customers can browse products, place orders, submit reviews, and even process returns.

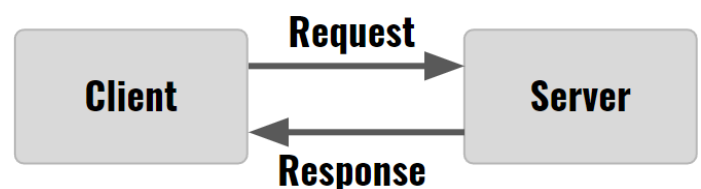


Creating visually appealing **user interfaces** is essential to capture customer attention, encourage purchases through marketing ads, and build long-term relationships through incentives like discounts and loyalty programs. User interfaces, as the name suggests, focus on application aspects that interact with users through visually engaging representations of data. Up to this point, these interfaces have used hard-coded JSON files, such as **courses.json** and **modules.json**, to render data. Interfaces have been built to render and manipulate this data, updating the screen to reflect changes.

However, these updates have not been permanent; refreshing the browser results in lost changes and a reset application state. JavaScript applications running on clients like browsers, game consoles, or TV boxes have limited options for retrieving and storing data permanently. The next chapters address the challenges of retrieving, storing, updating, and deleting data permanently on remote servers and databases from React.js applications.

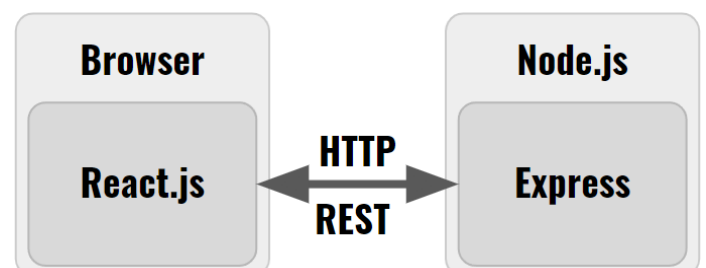
## 2 Installing and configuring an HTTP Web server

The Kanbas React Web application built so far is the **client** in a **client/server architecture**. Users interact with client applications that implement user interfaces relying on servers to store data and execute complex logic that would be impractical on the client. Clients and servers interact through a sequence of requests and responses. Clients send **requests** to servers, servers execute some logic, fulfill the request, and then **respond** back to the client with results. This section discusses implementing HTTP servers using Node.js.



### 2.1 Introduction to Node.js

JavaScript is generally recognized as a programming language designed to execute in browsers, but it has been rescued from its browser confines by Node.js. Node.js is a JavaScript runtime to interpret and execute applications written in JavaScript outside of browsers, such as from a desktop console or terminal. This allows JavaScript applications written for the desktop to overcome many limitations faced by those in the



browser. JavaScript running in a browser is restricted, with no access to the filesystem or databases, and limited network capabilities. In contrast, JavaScript running on a desktop has full access to the filesystem, databases, and unrestricted network access. Conversely, desktop JavaScript applications generally lack a user interface and offer limited user interaction, while browser-based JavaScript applications provide rich and sophisticated interfaces for user interaction.

## 2.2 Installing Node.js

**Node.js** is a JavaScript runtime that can execute JavaScript on a desktop, allowing JavaScript programs to breakout from the confines and limitations of a browser. Node.js was installed during previous assignments while implementing the React.js Web application. Confirm the installation and check the version by typing the following in your computer terminal or console application.

```
$ node -v  
v22.11.0
```

If a Node installation is present, its version will be displayed on the console; otherwise, an error message will be shown, indicating that Node.js needs to be downloaded and installed from the URL below. As of this writing, Node.js 22.11.0 was the latest version, but any version recommended on Node's website can be installed.

```
https://nodejs.org/en
```

Once downloaded, double click on the downloaded file to execute the installer, give the operating system all the permissions it requests, accept all the defaults, let the installer complete, and restart the computer. Once the computer is up and running again, confirm Node.js installed properly by running the command **node -v** again from the command line.

## 2.3 Creating a Node.js project

Another tool installed along with Node.js is **npm** or **Node Package Manager**. We've been using **npm** to run React applications in previous assignments. The **npm** command can be used to accomplish many more tasks, but like the name suggests, its main purpose is to install packages or executable code that **npm** can download and install in the local computer. Another important purpose of **npm** is to create brand new Node.js projects. To create a Node.js project create a directory with the name of the desired project and then change into that directory as shown below. Choose a directory name that does not contain any spaces, is all lowercase, and uses dashes between words.

Another tool that is installed along with Node.js is **npm**, or **Node Package Manager**. **Npm** has been used to run React applications in previous assignments. Many more tasks can be accomplished using the **npm** command, but as the name suggests, its main purpose is to install packages or executable code that can be downloaded and installed on the local computer. Another important purpose of **npm** is the creation of brand new Node.js projects. To create a Node.js project, a create directory with the name of the desired project, and then navigate into that directory, as shown below. Choose a directory name that does not contain any spaces, is all lowercase, and uses dashes between words.

```
$ mkdir kanbas-node-server-app  
$ cd kanbas-node-server-app
```

Once in the directory, use **npm init** to create a new Node.js project as shown below. This will kickoff an interactive session asking details about the project such as the name of the project and the author. The following is a sample interaction with sample answers. Each question provides a default answer which can be accepted or skipped by just pressing enter. It is fine to initially keep all the default values since they can be configured at a later time.

```
$ npm init
package name: (kanbas-node-server-app)
version: (1.0.0)
description: Node.js HTTP Web server for the Kanbas application
entry point: (index.js)
test command:
git repository: https://github.com/jannunzi/kanbas-node-server-app
keywords: Node, REST, Server, HTTP, Web Development, CS5610, CS4550
author: Jose Annunziato
license: (ISC)
```

The configuration will be written into a new file called **package.json** in the JSON format and it's distinctive of Node.js projects, like **pom.xml** files might be distinctive for Java projects.

## 2.4 Creating a Simple Hello World Node.js program

Open the project created earlier with an IDE such as [Visual Studio Code](#) or [IntelliJ](#), and at the root of the project, create a JavaScript file called **Hello.js** with the content shown below. The script uses the **console.log()** function to print the string **'Hello World!'** to the console and it is a common first program to write when learning a new language or infrastructure.

*Hello.js*

```
console.log("Hello World!");
```

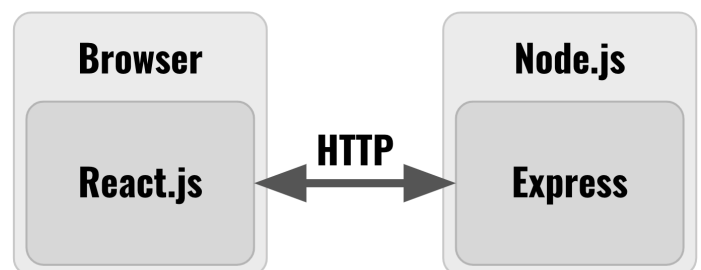
At the command line, run the **Hello.js** application by using the **node** command and confirm the application prints **Hello World!** to the console as shown below.

```
$ node Hello.js
Hello World!
```

Node.js programs consist of JavaScript files that are executed with the **node** command line interpreter. The following sections describe writing JavaScript applications that implement **HTTP Web servers** and **RESTful Web APIs** to integrate with React.js user interfaces. Upcoming assignments describe writing JavaScript applications that store and retrieve data from databases such as **MongoDB**.

## 2.5 Creating a Node.js HTTP Web server

**Express** is one of the most popular Node.js libraries that simplify creating HTTP servers. Express will be used to implement HTTP servers that can respond to HTTP requests from any HTTP client such as the React.js client implemented in earlier assignments. From the root directory of the Node.js project, install the **express** library from the terminal as shown below.



```
$ npm install express
```

Confirm that an **express** entry appears in **package.json** in the **dependencies** property. It is important these dependencies are listed in **package.json** so that they can be re-installed by other colleagues or when deploying to remote servers and cloud platforms such as **AWS**, **Heroku**, or **Render.js**. New libraries are installed in a new folder called **node\_modules**. More Node.js packages can be found at [npmjs.com](https://www.npmjs.com). The following **index.js** implements an HTTP server that responds **Hello World!** when the server receives an HTTP request at the URL <http://localhost:4000/hello>. Copy and paste the URL in a

browser to send the HTTP request and the browser will render the response from the server. The **require** function is equivalent to the **import** keyword and loads a library into the local source. The **express()** function call creates an instance of the express library and assigns it to local constant **app**. The **app** instance is used to configure the server on what to do when various types of requests are received. For instance the example below uses the **app.get()** function to configure an **HTTP GET Request handler** by mapping the URL pattern **'/hello'** to a function that handles the HTTP request.

*index.js*

```
const express = require('express')
const app = express()
app.get('/hello', (req, res) => {res.send('Hello World!')})
app.listen(4000)
```

```
// equivalent to import
// create new express instance
// create a route that responds 'hello'
// listen to http://localhost:4000
```

A request to URL <http://localhost:4000/hello> triggers the function implemented in the second argument of **app.get()**. The handler function receives parameters **req** and **res** which allows the function to participate in the **request/response** interaction, common in **client/server** applications. The **res.send()** function responds to the request with the text **Hello World!** Use **node** to run the server from the root of the project as shown below.

```
$ node index.js
```

The application will run, start the server, and wait at port **4000** for incoming HTTP requests. Point your browser to <http://localhost:4000/hello> and confirm the server responds with **Hello World!** Stop the server by pressing **Ctrl+C**. The string <http://localhost:4000/hello> is referred to as a **URL (Uniform Resource Locator)** and is used to .

## 2.6 Configuring Nodemon

React Web applications automatically transpile and restart every time code changes. Node.js can be configured to behave the same way by installing a tool called **nodemon** which monitors file changes and automatically restarts the Node application. Install nodemon globally (**-g**) as follows.

```
$ npm install nodemon -g
```

On **macOS** you might need to run the command as a **super user** as follows. Type your password when prompted.

```
$ sudo npm install nodemon -g
```

Now instead of using the node command to start the server, use **nodemon** as follows:

```
$ nodemon index.js
```

Confirm the server is still responding **Hello World!**. Change the response string to **Life is good!** and without stopping and restarting the server, refresh the browser and confirm that the server now responds with the new string. To practice, create another endpoint mapped to the root of the application, e.g., **"/**. Navigate to <http://localhost:4000> with your browser and confirm the server responds with the message below.

*index.js*

```
const express = require('express')
const app = express()
app.get('/hello', (req, res) => {res.send('Life is good!')})
app.get('/', (req, res) => {
  res.send('Welcome to Full Stack Development!')})
app.listen(4000)
```

```
// http://localhost:4000/hello responds "Life is good"
// http://localhost:4000 responds "Welcome to Full ..."
```

## 2.7 Configuring Node.js to use ES6

So far we've been using the keyword **import** to load ES6 modules in our React Web applications, but in **index.js** we used **require** instead to accomplish the same thing. Since Node version 12, ES6 syntax is supported by configuring the **package.json** file and adding a new **"type"** property with value **"module"** as shown below in the highlighted text.

*package.json*

```
{
  "type": "module",
  "name": "kanbas-node-server-app",
  ...
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node index.js"
  },
  // type module turns on ES6
  // use npm start to start server
```

Now, instead of using **require()** to load libraries, the familiar **import** statement can be used instead. Here's the **index.js** refactored to use **import** instead of **require**. Restart the server, refresh the browser and confirm that the server responds as expected.

*index.js*

```
import express from 'express';
const app = express();
app.get('/hello', (req, res) => {res.send('Life is good!')})
app.get('/', (req, res) => {res.send('Welcome to Full Stack Development!')})
app.listen(4000);
```

## 2.8 Creating HTTP Routes

The **index.js** file creates and configures an HTTP server listening for incoming HTTP requests. So far we've created a simple **hello** HTTP **route** that responds with a simple string. Throughout this and later assignments, we're going to create quite a few other HTTP routes, too many to define them all in **index.js**. Move both HTTP routes to the **Hello.js** file created earlier as shown below.

*index.js*

```
import express from 'express';
const app = express();
app.get('/hello', (req, res) => {
  res.send('Life is good!')
});
app.get('/', (req, res) => {
  res.send('Welcome to Full Stack Development!')
});
app.listen(4000);
// move this to Hello.js
```

Copy the routes to **Hello.js** as shown below.

*Hello.js*

```
console.log('Hello world!');
app.get('/hello', (req, res) => {
  res.send('Life is good!')
});
app.get('/', (req, res) => {
  res.send('Welcome to Full Stack Development!')
});
// don't need anymore
// moved here from index.js
```

In our case **Hello.js** handles HTTP requests for a **hello** greeting and responds with a friendly reply. We're not done though. Notice that **Hello.js** references **app** which is undefined in the file. Let's pass **app** as a parameter in a function we can import and invoke from **index.js** as shown below. Test **http://localhost:4000/hello** from the browser and confirm the reply is still friendly.

#### Hello.js

```
export default function Hello(app) {  
  app.get('/hello', (req, res) => {  
    res.send('Life is good!')  
  })  
  app.get('/', (req, res) => {  
    res.send('Welcome to Full Stack Development!')  
  })  
}
```

*// function accepts app reference to express module  
// to create routes here. We could have used the new  
// arrow function syntax instead*

Import **Hello.js** and pass **app** to the function as shown below.

#### index.js

```
import express from 'express'  
import Hello from './Hello.js'  
const app = express()  
Hello(app)  
app.listen(4000)
```

*// import Hello from Hello.js  
// pass app reference to Hello*

## 3 Labs

The following are a set of exercises to practice creating and integrating with an HTTP server from a React.js Web application. In your server application, create and import file **Lab5/index.js** where we will be implementing several server side exercises. Create a route that welcomes users to Lab 5.

#### Lab5/index.js

```
export default function Lab5(app) {  
  app.get("/lab5/welcome", (req, res) => {  
    res.send("Welcome to Lab 5");  
  });  
};
```

*// accept app reference to express module  
// create route to welcome users to Lab 5.  
// Here we are using the new arrow function syntax*

Import **Lab5/index.js** into the server **index.js** and pass it a reference to **express** as shown below. Restart the server and confirm that <http://localhost:4000/lab5/welcome> responds with the expected response.

#### index.js

```
import Hello from './Hello.js';  
import Lab5 from './Lab5/index.js';  
const app = express();  
app.use(express.json());  
Lab5(app);  
Hello(app);
```

*// import Lab5  
// pass reference to express module*

In the React Web app project, create a **Lab5** React component to test the Node HTTP server. Import the new component into the existing set of labs and add a new link in **TOC.tsx** to navigate to **Lab5** by selecting the corresponding tab or link as shown here on the right. The example below creates a hyperlink that navigates to the <http://localhost:4000/lab5/welcome> URL. Confirm the link navigates to the expected response.

[Lab 1](#)[Lab 2](#)[Lab 3](#)[Lab 4](#)[Lab 5](#)

## Lab 5

Welcome

src/Labs/Lab5/index.tsx

```
export default function Lab5() {
  return (
    <div id="wd-lab5">
      <h2>Lab 5</h2>
      <div className="list-group">
        <a href="http://localhost:4000/lab5/welcome" // hyperlink navigates to
          className="list-group-item" // http://localhost:4000/lab5/welcome
        >Welcome
        </a>
      </div><hr/>
    </div>
  );
}
```

## 3.1 Environment Variables

Currently we are integrating the React user interface with a Node server that are both running locally on our development computers, but ultimately these will both be running on remote servers. Let's configure the local environment so that it will be easy later on to configure our source code to run in any environment. In this course we will be concerned about two environments: our **local development environment** and the **remote production environment**. For us the **local development environment** consists of our computer where we do our development running two Node servers, one hosting the React user interface and the other hosting the Express HTTP server. The **remote production environments** will consist of the React user interface running on Netlify, and the Express HTTP server running on **Render.com**, **Heroku**, or **AWS** (your choice). Environments can be configured with **environment variables** declared in your **Operating System** or as **environment files** in your project. Environment files are named **.env** (with a leading period) and can be defined for each environment by appending **.local** for the local environment, **.test** for the test environment, and **.production** for the production. Instead of using **http://localhost:4000** everywhere in our source code, let's declare an environment variable in the local environment file as shown below. Create the **.env.local** file at the root of your React project.

.env.local

```
REACT_APP_REMOTE_SERVER=http://localhost:4000
```

All environment variables must start with **REACT\_APP\_**. Each line declares a variable, followed by an equal sign, followed by the value of the variable. Do not use extra spaces or unnecessary extra characters such as quotes, commas, or colons. Everytime you add, remove, or make changes to an environment file, the React user interface application needs to be restarted. Environment variables can be accessed from the React source code through the global **process** object in its **env** property. To instance, to access the value of the **REACT\_APP\_REMOTE\_SERVER** declared above, use **process.env.REACT\_APP\_REMOTE\_SERVER**. To practice declaring and using environment variables, create component **EnvironmentVariables** as shown below.

src/Labs/Lab5/EnvironmentVariables.tsx

```
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
export default function EnvironmentVariables() {
  return (
    <div id="wd-environment-variables">
      <h3>Environment Variables</h3>
      <p>Remote Server: {REMOTE_SERVER}</p><hr/>
    </div>
  );
}
```

Import the component in component **Lab5**, restart the React application and confirm the URL of the remote server displays as shown below.



src/Labs/Lab5/index.tsx

```
import EnvironmentVariables from "../EnvironmentVariables";
export default function Lab5() {
  return (
    <div id="wd-lab5">
      <h2>Lab 5</h2>
      <div className="list-group">
        <a href="http://localhost:4000/lab5/welcome"
          className="list-group-item">
          Welcome
        </a>
      </div><hr />
      <EnvironmentVariables />
    </div>
  );
}
```

## Lab 5

Welcome

## Environment Variables

Remote Server: http://localhost:4000

Make sure the URL to the remote server is never used in the React source code, instead prefer using the environment variable. Replace the **http://localhost:4000** in the previous exercise with the **REMOTE\_SERVER** constant as shown below. Confirm that the **Welcome** hyperlink still works.

src/Labs/Lab5/index.tsx

```
import EnvironmentVariables from "../EnvironmentVariables";
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
export default function Lab5() {
  return (
    <div id="wd-lab5">
      <h2>Lab 5</h2>
      <div className="list-group">
        <a href={` ${REMOTE_SERVER}/lab5/welcome` } className="list-group-item">
          Welcome
        </a>
      </div><hr />
      <EnvironmentVariables />
    </div>
  );
}
```

Similarly, the Node Express server needs to be configured to run locally on your computer as well as when it is deployed in the remote environment. To do this, refactor **index.js** so that it uses the remote **PORT** environment variable if available, or port 4000 when running locally

index.js

```
import express from 'express'
import Hello from "../Hello.js"
import Lab5 from "../Lab5/index.js";
const app = express()
Lab5(app)
Hello(app)
app.listen(process.env.PORT || 4000)
```

## 3.2 Sending Data to a Server via HTTP Requests

Let's explore how we can integrate the React.js user interface with the Node server by sending information to the server from the browser. There are three ways to send information to the server:

1. **Path parameters** - parameters are encoded as segments of the path itself, e.g., **http://localhost:4000/lab5/add/2/5**.
2. **Query parameters** - parameters are encoded as name value pairs in the query string after the ? character at the end of a URL, e.g., **http://localhost:4000/lab5/add?a=2&b=5**.



3. **Request body** - data is sent as a string representation of data encoded in some format such as XML or JSON containing properties and their values, e.g., `{a:2, b: 5}` or `<params a=2 b=5/>`.

We'll explore the first two in this section, and address the last one towards the end of the labs.

## 3.2.1 Sending Data to a Server with Path Parameters

React.js applications can pass data to servers by embedding it in a URL path as **path parameters** part of a URL. For instance the last two integers – 2 and 4 – at the end of the following URL can be parsed by a corresponding matching route on the server, add the two integers, and respond with the result of 6.

<http://localhost:4000/lab5/add/2/4>

The following route declarations can parse path parameters `a` and `b` encoded in paths `/lab5/add/:a/:b` and `/lab5/subtract/:a/:b`. In **PathParameters.js**, implement the routes below and import it in **Lab5/index.js**. On your own create routes `/lab5/multiply/:a/:b` and `lab5/divide/:a/:b` that calculate the arithmetic multiplication and division.

**Lab5/PathParameters.js**

```
export default function PathParameters(app) {
  app.get("/lab5/add/:a/:b", (req, res) => {
    const { a, b } = req.params;
    const sum = parseInt(a) + parseInt(b);
    res.send(sum.toString());
  });
  app.get("/lab5/subtract/:a/:b", (req, res) => {
    const { a, b } = req.params;
    const sum = parseInt(a) - parseInt(b);
    res.send(sum.toString());
  });
};
```

*// route expects 2 path parameters after /Lab5/add  
// retrieve path parameters as strings  
// parse as integers and adds  
// sum as string sent back as response  
// don't send integers since can be interpreted as status  
// route expects 2 path parameters after /Lab5/subtract  
// retrieve path parameters as strings  
// parse as integers and subtracts  
// subtraction as string sent back as response  
// response is converted to string otherwise browser  
// would interpret integer response as a status code*

Note when you import, make sure to include the extension `.js` as shown below. Pass a reference of **app** to the **PathParameters** function. Confirm that <http://localhost:4000/lab5/add/6/4> responds with 10 and <http://localhost:4000/lab5/subtract/6/4> responds with 2. Also confirm the routes you implemented on your own.

**Lab5/index.js**

```
import PathParameters from "./PathParameters.js";
export default function Lab5(app) {
  app.get("/lab5/welcome", (req, res) => {
    res.send("Welcome to Lab 5");
  });
  PathParameters(app);
}
```

Meanwhile, in the React.js Web application, let's create a React.js component to test the new routes from our Web application. Web applications that interact with server applications are often referred to as **client applications** since they are the **client** in an application built using a **client server architecture**. Create the component below that declares state variables `a` and `b`, encodes the values in hyperlinks, and when you click them, server responds with the addition or subtraction of the parameters. Note that the name of the component is arbitrary. The fact that it is called the same as the routes in the server is a coincidence. Also helps us keep track of which UI components on the client are related to the server resources. On your own, create links that invoke the multiply and divide routes you implemented earlier. Import the new component in your **Lab5** component and confirm that clicking the links generates the expected response. Confirm all the routes work when you click the links in the browser.

### Path Parameters

Add 34 + 23Subtract 34 - 23

src/Labs/Lab5/PathParameters.tsx

```
import React, { useState } from "react";
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
export default function PathParameters() {
  const [a, setA] = useState("34");
  const [b, setB] = useState("23");
  return (
    <div>
      <h3>Path Parameters</h3>
      <input className="form-control mb-2" id="wd-path-parameter-a" type="number" defaultValue={a}
        onChange={(e) => setA(e.target.value)} />
      <input className="form-control mb-2" id="wd-path-parameter-b" type="number" defaultValue={b}
        onChange={(e) => setB(e.target.value)} />
      <a className="btn btn-primary me-2" id="wd-path-parameter-add"
        href={` ${REMOTE_SERVER}/lab5/add/${a}/${b}` }>
        Add {a} + {b}
      </a>
      <a className="btn btn-danger" id="wd-path-parameter-subtract"
        href={` ${REMOTE_SERVER}/lab5/subtract/${a}/${b}` }>
        Subtract {a} - {b}
      </a>
      <hr />
    </div>
  );
}
```

### 3.2.2 Sending Data to a Server with Query Parameters

### Query Parameters

React.js applications can also send data to servers by encoding it as a **query string** after the **question mark** character (?) at the end of a URL. A **query string** consists of a list of name value pairs separated by the **ampersand** character (&) as shown below.

<http://localhost:4000/lab5/calculator?operation=add&a=2&b=4>

34

23

Add 34 + 23

Subtract 34 - 23

In **Lab5/QueryParameters.js**, in your server application, create a route that can parse an **operation** and its parameters **a** and **b** as shown below. If the **operation** is **add** the route responds with the addition of the parameters. If the **operation** is **subtract**, the route responds with the subtraction of the parameters. On your own, also handle operations **multiply** and **divide**. Import **QueryParameters.js** in **Lab5/index.js** and pass it a reference of **app** (not shown).

Lab5/QueryParameters.js

```
export default function QueryParameters(app) {
  app.get("/lab5/calculator", (req, res) => {
    const { a, b, operation } = req.query;
    let result = 0;
    switch (operation) {
      case "add":
        result = parseInt(a) + parseInt(b);
        break;
      case "subtract":
        result = parseInt(a) - parseInt(b);
        break;
      // implement multiply and divide on your own
      default:
        result = "Invalid operation";
    }
    res.send(result.toString());
  });
}
```

// e.g., Lab5/calculator?a=5&b=2&operation=add  
// retrieve a, b, and operation parameters in query

// parse as integers since parameters are strings

// convert to string otherwise browser interprets  
// as a status code

In a new component **QueryParameters.tsx**, in your React.js Web application, create hyperlinks to test the new route as shown below. You'll need to declare **const removeServer** initialized to the environment variable **REACT\_APP\_REMOTE\_SERVER**. Confirm the following hyperlinks work as expected.

Test Hyperlinks	Confirm Response
<a href="http://localhost:4000/lab5/calculator?operation=add&amp;a=34&amp;b=23">http://localhost:4000/lab5/calculator?operation=add&amp;a=34&amp;b=23</a>	57
<a href="http://localhost:4000/lab5/calculator?operation=subtract&amp;a=34&amp;b=23">http://localhost:4000/lab5/calculator?operation=subtract&amp;a=34&amp;b=23</a>	11

*src/Labs/Lab5/QueryParameters.tsx*

```
<div id="wd-query-parameters">
  <h3>Query Parameters</h3>
  <input id="wd-query-parameter-a"
    className="form-control mb-2"
    defaultValue={a} type="number"
    onChange={(e) => setA(e.target.value)} />
  <input id="wd-query-parameter-b"
    className="form-control mb-2"
    defaultValue={b} type="number"
    onChange={(e) => setB(e.target.value)} />
  <a id="wd-query-parameter-add"
    href={` ${REMOTE_SERVER}/lab5/calculator?operation=add&a=${a}&b=${b}`} >
    Add {a} + {b}
  </a>
  <a id="wd-query-parameter-subtract"
    href={` ${REMOTE_SERVER}/lab5/calculator?operation=subtract&a=${a}&b=${b}`} >
    Subtract {a} - {b}
  </a>
  /* create additional links to test multiply and divide. use IDs starting with wd-query-parameter- */
<hr />
</div>
```

## 3.2.3 On Your Own

On your own, remember to implement **multiply** and **divide** requests on the client and server that demonstrate multiplying and dividing numbers **encoded in the request's path**. Now implement the same operations again, **multiply** and **divide** on the server and client that demonstrate, but multiplying and dividing parameters **encoded in the query string**.

## 3.3 Working with Remote Objects on a Server

The examples so far have demonstrated working with integers and strings, but all primitive datatypes work as well, including objects and arrays. The example below declares an **assignment** object accessible at the route **/lab5/assignment**. Import it to your **Lab5/index.js** in your server.

*Lab5/WorkingWithObjects.js*

```
const assignment = {
  id: 1, title: "NodeJS Assignment",
  description: "Create a NodeJS server with ExpressJS",
  due: "2021-10-10", completed: false, score: 0,
};
export default function WorkingWithObjects(app) {
  app.get("/lab5/assignment", (req, res) => {
    res.json(assignment);
  });
};

// object state persists as long
// as server is running
// changes to the object persist
// rebooting server
// resets the object

// use .json() instead of .send() if you know
// the response is formatted as JSON
```

### 3.3.1 Retrieving Objects from a Server

In your React.js project, create a **WorkingWithObjects** component to test the new route as shown below. Import it in **Lab5** and confirm that <http://localhost:4000/lab5/assignment> responds with **assignment** object.

src/Labs/Lab5/WorkingWithObjects.tsx

```
import React, { useState } from "react";
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
export default function WorkingWithObjects() {
  return (
    <div id="wd-working-with-objects">
      <h3>Working With Objects</h3>
      <h4>Retrieving Objects</h4>
      <a id="wd-retrieve-assignments" className="btn btn-primary"
        href={` ${REMOTE_SERVER}/lab5/assignment`} >
        Get Assignment
      </a><hr/>
    </div>
  );
};
```

## Working With Objects

### Retrieving Objects

Get Assignment

### 3.3.2 Retrieving Object Properties from a Server

We can retrieve individual properties in an object such as the **title** shown below.

Lab5/index.js

```
export default function WorkingWithObjects(app) {
  app.get("/lab5/assignment", (req, res) => {
    res.json(assignment);
  });
  app.get("/lab5/assignment/title", (req, res) => {
    res.json(assignment.title);
  });
}
```

## Retrieving Properties

Get Title

Confirm that the <http://localhost:4000/lab5/assignment/title> hyperlink below retrieves the assignment's title. In **WorkingWithObjects**, add a link that retrieves the title as shown below. Confirm that clicking the link retrieves the assignment's title.

src/Labs/Lab5/WorkingWithObjects.tsx

```
...
<h4>Retrieving Objects</h4>
<a id="wd-retrieve-assignments" className="btn btn-primary"
  href={` ${REMOTE_SERVER}/lab5/assignment`} >
  Get Assignment
</a><hr/>
<h4>Retrieving Properties</h4>
<a id="wd-retrieve-assignment-title" className="btn btn-primary"
  href={` ${REMOTE_SERVER}/lab5/assignment/title`} >
  Get Title
</a><hr/>
...
```

### 3.3.3 Modifying Objects in a Server

We can also use routes to modify objects or individual properties as shown below. The route retrieves the new title from the path and updates the **assignment**'s object's **title** property.

Lab5/WorkingWithObjects.js

```
app.get("/lab5/assignment/title/:newTitle", (req, res) => {
  const { newTitle } = req.params;
  assignment.title = newTitle;
  res.json(assignment);
});
```

// changes to objects in the server  
// persist as long as the server is running  
// rebooting the server resets the object state

In **WorkingWithObjects** component in your React.js project, create an **assignment** state variable to test editing the **assignment** object on the server. Create an input field where we can type the new assignment title, and a link that invokes the route that updates the title. Confirm that you can change the assignment's title.

## Modifying Properties

src/Labs/Lab5/WorkingWithObjects.tsx

```
import React, { useState } from "react";
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
export default function WorkingWithObjects() {
  const [assignment, setAssignment] = useState({
    id: 1, title: "NodeJS Assignment",
    description: "Create a NodeJS server with ExpressJS",
    due: "2021-10-10", completed: false, score: 0,
  });
  const ASSIGNMENT_API_URL = `${REMOTE_SERVER}/lab5/assignment`
  return (
    <div>
      <h3 id="wd-working-with-objects">Working With Objects</h3>
      <h4>Modifying Properties</h4>
      <a id="wd-update-assignment-title"
        className="btn btn-primary float-end"
        href={` ${ASSIGNMENT_API_URL}/title/${assignment.title}`}>
        Update Title
      </a>
      <input className="form-control w-75" id="wd-assignment-title"
        defaultValue={assignment.title} onChange={(e) =>
          setAssignment({ ...assignment, title: e.target.value })/>
      <hr />
      ...
    </div>
  );
};
```

NodeJS Assignment

Update Title

```
// create a state variable that holds
// default values for the form below.
// eventually we'll fetch this initial
// data from the server and populate
// the form with the remote data so
// we can modify it here in the UI
```

```
// encode the title in the URL that
// updates the title
```

```
// form element to edit local state variable
// used to encode in URL that updates
// property in remote object
```

### 3.3.4 On Your Own

Now, on your own, create a **module** object with **string** properties **id**, **name**, **description**, and **course**. Feel free to use values of your choice.

- Create a route that responds with the **module** object, mapped to **/lab5/module**
- In the UI, create a link labeled **Get Module** that retrieves the **module** object from the server mapped at **/lab5/module**
- Confirm that clicking the link retrieves the module.
- Create another route mapped to **/lab5/module/name** that retrieves the **name** of the **module** created earlier
- In the UI, create a hyperlink labeled **Get Module Name** that retrieves the **name** of the **module** object
- Confirm that clicking the link retrieves the module's name.

On your own, in **WorkingWithObjects.tsx**, create a **module** state variable to test editing the **module** object on the server. Create an input field where we can type the new module name, and a link that invokes the route that updates the name. Confirm that you can change the module's name. Create routes and a corresponding UI that can modify the **score** and **completed** properties of the **assignment** object. In the React application, create an input field of type **number** where you can type the new **score** and an input field of type **checkbox** where you can select the **completed** property. Create a link that updates the **score** and another link that updates the **completed** property. For the module, create routes and UI to edit the module's description.

## 3.4 Working with Remote Arrays on a Server

Now let's work with something a little more challenging. Create an array of objects and explore how to retrieve, add, remove, and update the array. Working with a collection of objects requires a general set of operations often referred to as **CRUD** or **create**, **read**, **update**, and **delete**. These operations capture common interactions with any collection of data such

as **creating** and adding new instances to the collection, **reading** or retrieving items in a collection, **updating** or modifying items in a collection, and **deleting** or removing items from a collection.

### 3.4.1 Retrieving Arrays from a Server

Let's First create the array and create the route below to retrieve the array. Import the new route to **Lab5/index.js**.

*Lab5/WorkingWithArrays.js*

```
let todos = [ { id: 1, title: "Task 1", completed: false }, { id: 2, title: "Task 2", completed: true },
               { id: 3, title: "Task 3", completed: false }, { id: 4, title: "Task 4", completed: true }, ];
export default function WorkingWithArrays(app) {
  app.get("/lab5/todos", (req, res) => {
    res.json(todos);
  });
};
```

In a new React component, create a hyperlink to test retrieving the array. Add the new component to the **Lab5** complement and confirm clicking the link retrieves the array.

*src/Labs/Lab5/WorkingWithArrays.tsx*

```
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
export default function WorkingWithArrays() {
  const API = `${REMOTE_SERVER}/lab5/todos`;
  return (
    <div id="wd-working-with-arrays">
      <h3>Working with Arrays</h3>
      <h4>Retrieving Arrays</h4>
      <a id="wd-retrieve-todos" className="btn btn-primary" href={API}>
        Get Todos </a><hr/>
      </div>
    );
};
```

## Retrieving Arrays

Get Todos

### 3.4.2 Retrieving Data from a Server by its Primary Key

Another common operations is to retrieve a particular item from an array by its primary key, e.g., its ID property. The convention is to encode the ID of the item of interest as a path parameter. The example below parses the ID from the path parameter, finds the corresponding item, and responds with the item.

*Lab5/WorkingWithArrays.js*

```
app.get("/lab5/todos/:id", (req, res) => {
  const { id } = req.params;
  const todo = todos.find((t) => t.id === parseInt(id));
  res.json(todo);
});
```

Add a hyperlink to the React component to test retrieving an item from the array by its primary key. Confirm that you can type the ID in the UI and clicking the hyperlink retrieves the corresponding item.

*src/Labs/Lab5/WorkingWithArrays.tsx*

```
import React, { useState } from "react";
export default function WorkingWithArrays() {
  ...
  const [todo, setTodo] = useState({id: "1"});
  ...
  <h4>Retrieving Arrays</h4>
  <a id="wd-retrieve-todos" className="btn btn-primary" href={API}>
    Get Todos
  </a><hr/>
}
```

## Retrieving an Item from an Array by ID

1

Get Todo by ID

```

<h4>Retrieving an Item from an Array by ID</h4>
<a id="wd-retrieve-todo-by-id" className="btn btn-primary float-end" href={` ${API}/${todo.id}`}>
  Get Todo by ID
</a>
<input id="wd-todo-id" defaultValue={todo.id} className="form-control w-50"
  onChange={(e) => setTodo({ ...todo, id: e.target.value })} />
<hr />
...

```

### 3.4.3 Filtering Data from a Server with a Query String

The convention for retrieving a particular item from a collection is to encode the item's ID as a path parameter, e.g., `/todos/123`. Another convention is that if the primary key is not provided, then the interpretation is that we want the entire collection of items, e.g., `/todos`. We can also want to retrieve items by some other criteria other than the item's ID such as the item's **title** or **completed** properties. The convention in this case is to use query strings instead of path parameters when filtering items by properties other than the primary key, e.g., `/todos?completed=true`. The example below refactors the `/lab5/todos` to handle the case when we want to filter the array by the **completed** query parameter.

*Lab5/WorkingWithArrays.js*

```

const todos = [ { id: 1, title: "Task 1", completed: false }, { id: 2, title: "Task 2", completed: true },
  { id: 3, title: "Task 3", completed: false }, { id: 4, title: "Task 4", completed: true }, ];
export default function WorkingWithArrays(app) {
  app.get("/lab5/todos", (req, res) => {
    const { completed } = req.query;
    if (completed !== undefined) {
      const completedBool = completed === "true";
      const completedTodos = todos.filter(
        (t) => t.completed === completedBool);
      res.json(completedTodos);
      return;
    }
    res.json(todos);
  });
};

```

Add a hyperlink to the React component to test retrieving all completed todos.

*src/Labs/Lab5/WorkingWithArrays.tsx*

```

...
<h3>Retrieving Arrays</h3>
...
<h3>Filtering Array Items</h3>
<a id="wd-retrieve-completed-todos" className="btn btn-primary"
  href={` ${API}?completed=true`} >
  Get Completed Todos
</a><hr/>
...

```

## Filtering Array Items

Get Completed Todos

### 3.4.4 Creating New Data in a Server

The examples we've seen so far have illustrated various **read** operations in our exploration of possible **CRUD** operations. Let's now take a look at the **create** operation. The example below demonstrates a route that creates a new item in the array and responds with the array now containing the new item. Note that it is implemented before the `/lab5/todos/:id` route, otherwise the `:id` path parameter would interpret the **"create"** in `/lab5/todos/create` as an ID, which would certainly create an error trying to parse it as an integer. Also note that the `newTodo` creates default values including a unique identifier field `id` based on a timestamp. Eventually primary keys will be handled by a database later in the course. Finally note that the response consists of the entire `todos` array, which is convenient for us for now, but a more common implementation would be to respond with `newTodo`.



Lab5/WorkingWithArrays.js

```
export default function WorkingWithArrays(app) {
  app.get("/lab5/todos/create", (req, res) => {
    const newTodo = {
      id: new Date().getTime(),
      title: "New Task",
      completed: false,
    };
    todos.push(newTodo);
    res.json(todos);
  });
  app.get("/lab5/todos/:id", (req, res) => {
    const { id } = req.params;
    const todo = todos.find((t) => t.id === parseInt(id));
    res.json(todo);
  });
  ...
}
```

// make sure to implement this BEFORE the /lab5/todos/:id route implemented below

// make sure to implement this AFTER the /lab5/todos/create route implemented above

Add a **Create Todo** hyperlink to the **WorkingWithArrays** component to test the new route. Confirm that clicking the link creates the new item in the array.

src/Labs/Lab5/WorkingWithArrays.tsx

```
...
<h3>Creating new Items in an Array</h3>
<a id="wd-retrieve-completed-todos" className="btn btn-primary"
  href={` ${API}/create`} >
  Create Todo
</a><hr/>
...
```

### Creating new Items in an Array

Create Todo

## 3.4.5 Deleting Data from a Server

Next let's consider the **delete** operation in the **CRUD** family of operations. The convention is to encode the ID of the item to delete as a path parameter as shown below. We search for the item in the set of items and remove it. Typically we would respond with a status of success or failure, but for now we're responding with all the todos for now.

Lab5/WorkingWithArrays.js

```
...
app.get("/lab5/todos/:id/delete", (req, res) => {
  const { id } = req.params;
  const todoIndex = todos.findIndex((t) => t.id === parseInt(id));
  todos.splice(todoIndex, 1);
  res.json(todos);
});
...
```

### Deleting from an Array

Delete Todo with ID = 2

To test the new route, create a link that encodes the todo's ID in a hyperlink to delete the corresponding item. We'll use the **todo** state variable created earlier to type the ID of the item we want to remove. Confirm that you can type the ID of an item, click the hyperlink and that the corresponding item is removed from the array.

src/Labs/Lab5/WorkingWithArrays.tsx

```
...
<h3>Deleting from an Array</h3>
<a id="wd-retrieve-completed-todos" className="btn btn-primary float-end" href={` ${API}/${todo.id}/delete`} >
  Delete Todo with ID = {todo.id} </a>
<input defaultValue={todo.id} className="form-control w-50" onChange={(e) => setTodo({ ...todo, id: e.target.value
  })}/><hr/>
...
```

## 3.4.6 Updating Data on a Server

Finally let's consider the **update** operation in the **CRUD** family of operations. The convention is to encode the ID of the item to update as a path parameter as shown below. We search for the item in the set of items and update it. Typically we would respond with a status of success or failure, but for now we're responding with all the todos for now.

*Lab5/WorkingWithArrays.js*

```
...
app.get("/lab5/todos/:id/title/:title", (req, res) => {
  const { id, title } = req.params;
  const todo = todos.find((t) => t.id === parseInt(id));
  todo.title = title;
  res.json(todos);
});
...
```

To test the new route, add an input field to edit the **title** property and a link that encodes both the ID of the item and the new value of the **title** property as shown below

*src/Labs/Lab5/WorkingWithArrays.tsx*

```
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
export default function WorkingWithArrays() {
  const [todo, setTodo] = useState({
    id: "1",
    title: "NodeJS Assignment",
    description: "Create a NodeJS server with ExpressJS",
    due: "2021-09-09",
    completed: false,
  });
  return (
    <div>
      <h2>Working with Arrays</h2>
      ...
      <h3>Updating an Item in an Array</h3>
      <a href={` ${API}/${todo.id}/title/${todo.title}`} className="btn btn-primary float-end">
        Update Todo</a>
      <input defaultValue={todo.id} className="form-control w-25 float-start me-2"
        onChange={(e) => setTodo({ ...todo, id: e.target.value })}/>
      <input defaultValue={todo.title} className="form-control w-50 float-start"
        onChange={(e) => setTodo({ ...todo, title: e.target.value })}/>
      <br /><br /><hr />
    </div>
  );
}
```

### Updating an Item in an Array

1

NodeJS Assignment

Update Todo

## 3.4.7 On Your Own

Using the exercises so far as examples, implement routes and corresponding UI that allows editing a **completed** and **description** properties of **todo** items identified by their ID. Create the routes below in **Lab5/index.js** in the Node.js HTTP server project. In **WorkingWithArrays** component, add a text input field to edit the **description** and a checkbox input field to edit the **completed** property. Create a link that updates the **description** of the todo item whose **id** is encoded in the URL and another link that updates the **completed** property of the todo item whose **id** is encoded in the URL.

Property	Routes	Response	Test
completed	/lab5/todos/:id/completed/:completed	todos	<a href="#">Complete Todo ID = 1</a>
description	/lab5/todos/:id/description/:description	todos	<a href="#">Describe Todo ID = 1</a>

## 3.5 Asynchronous Communication with HTTP Servers

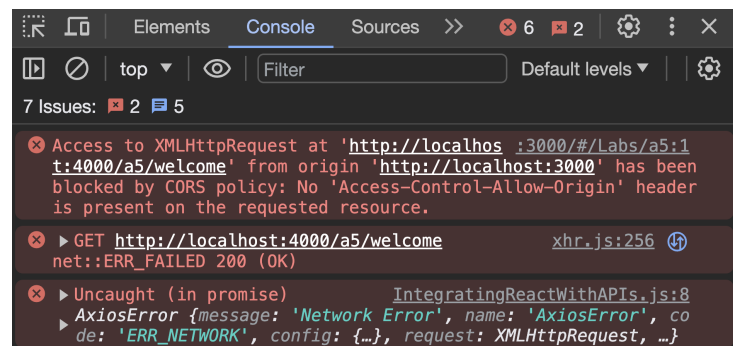
The exercises explored so far sent data encoded in the URL of hyperlinks. The links navigated to a separate browser window that displayed the server response. Even though we were able to send data to the server and affect changes to the server data, we have not considered how those changes can affect the user interface. Let's explore how to fully integrate the user interface with the server by sending and receiving HTTP requests and responses asynchronously.

### 3.5.1 Asynchronous JavaScript and XML

JavaScript Web applications, such as React.js applications, can communicate with server applications using a technology called **AJAX** or **Asynchronous JavaScript and XML**. Using **AJAX** JavaScript applications can send and retrieve HTTP requests and response asynchronously from client JavaScript applications to remote HTTP server. Although **XML** was the original data format in **AJAX**, **JSON** has overtaken as the dominant data format in modern Web applications, but the **AJAX** label still applies nevertheless. **Axios** is a popular JavaScript library that React.js user interface applications can use to communicate with servers using **AJAX**. Install the library at the root of the React.js Web application project as shown below.

```
$ npm install axios
```

Let's use the same server routes implemented in earlier exercises, but instead of clicking on hyperlinks in the React client, we'll use **axios** to programmatically invoke the URLs giving us a chance to capture and handle the responses from the server and render the response in the user interface. The code below illustrates how to use the **axios** library to send an asynchronous request to the server and then capture the response in the user interface, without navigating to the URL, away from the current window. The **fetchWelcomeOnClick** function is tagged as **async** since it uses **axios.get()** to asynchronously send a request to the server, and returns the response from the server. Create the component below and import it in the **Lab5** component. Open the **Web Dev Tools** and confirm that clicking the **Fetch Welcome** button causes the **CORS** error shown here on the right.



src/Labs/Lab5/HttpClient.tsx

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

const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;

export default function HttpClient() {
  const [welcomeOnClick, setWelcomeOnClick] = useState("");

  const fetchWelcomeOnClick = async () => {
    const response = await axios.get(`${REMOTE_SERVER}/lab5/welcome`);
    setWelcomeOnClick(response.data);
  };

  return (
    <div>
      <h3>HTTP Client</h3> <hr />
      <h4>Requesting on Click</h4>
      <button className="btn btn-primary me-2" onClick={fetchWelcomeOnClick}>
        Fetch Welcome
      </button> <br />
      Response from server: <b>{welcomeOnClick}</b>
    </div>
  );
}
```

## HTTP Client

### Requesting on Click

Fetch Welcome

Response from server:

## 3.5.2 Configuring Cross Origin Request Sharing (CORS)

Servers and browsers limit **JavaScript** programs to only be able to communicate with the servers from where they are downloaded from. Since our **React.js** application is running locally from **localhost:3000**, then they would only be able to communicate back to a server running on **localhost:3000**, but our server is running on **localhost:4000**, so when our **JavaScript** components try to communicate with **localhost:4000**, the browser considers a different domain as a security risk, stops the communication and throws a **CORS** exception. **CORS** stands for **Cross Origin Request Sharing**, which governs the policies and mechanisms of how various resources can be shared across different domains or **origins**. Browsers enforce **CORS** policies by first checking with the server if they are ok with receiving requests from different domains. If the server responds affirmatively, then browsers let the requests go through, otherwise they'll consider the attempt as a violation of **CORS** security policy, abort the request, and throw the exception. We can configure the **CORS** security policies by installing the **cors** Node.js library as shown below.

```
$ npm install cors
```

In **index.js**, import the **cors** library and configure it as shown below to allow all requests from any origin. We'll narrow down this policy in a later chapter. Restart the server and refresh the React.js application. Confirm that the user interface is able to retrieve the **Welcome to Lab 5** message from the server without errors.

**index.js**

```
import express from "express";
import Lab5 from "../Lab5/index.js";
import cors from "cors";
const app = express();
app.use(cors()); // make sure cors is used right after creating the app
Lab5(app); // express instance
app.listen(4000);
```

## 3.5.3 Creating a Client Library

The current **HttpClient.tsx** implementation makes a request to the server from the component itself using the **axios** library. In addition to **retrieving** (or **reading**) data from the server, there will be other **CRUD** operations to **create**, **update**, and **delete** needed to interact with the server. Instead of implementing these in a React component, it's better to implement these in a reusable **client library** that can be shared across several user interface components. Move the **axios.get()** in **HttpClient.tsx** to a separate file called **client.ts** as shown below.

**src/Labs/Lab5/client.ts**

```
import axios from "axios";
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
export const fetchWelcomeMessage = async () => {
  const response = await axios.get(`${REMOTE_SERVER}/lab5/welcome`);
  return response.data;
};
```

Now refactor **HelloClient.ts** to use the **client.ts** as shown below. Confirm that clicking on **Fetch Welcome** still works.

**src/Labs/Lab5/HttpClient.tsx**

```
import * as client from "../client";
export default function HttpClient() {
  const [welcomeOnClick, setWelcomeOnClick] = useState("");
  const fetchWelcomeOnClick = async () => {
    const message = await client.fetchWelcomeMessage();
    setWelcomeOnClick(message);
  };
  return ( ... );}
```

## 3.5.4 Retrieving Data from a Server on Component Load

The previous exercise fetched data from the server when the user requested it. Often times we need to retrieve data from the server when you first navigate to a screen or a component is first loaded and displayed. Use React's **useEffect** hook function as shown below to invoke the **fetchWelcomeOnLoad** when a component or screen first loads. Now, when the **HttpClient** loads, the **useEffect** invokes **fetchWelcomeOnLoad** which retrieves the message from the server and sets the new **welcomeOnLoad** state variable. Confirm that if you refresh the screen, the **welcome** message appears without having to click on the **Fetch Welcome** button.

src/Labs/Lab5/HttpClient.tsx

```
import React, { useEffect, useState } from "react";
import * as client from "../client";
export default function HttpClient() {
  const [welcomeOnClick, setWelcomeOnClick] = useState("");
  const [welcomeOnLoad, setWelcomeOnLoad] = useState("");
  const fetchWelcomeOnClick = async () => {
    const message = await client.fetchWelcomeMessage();
    setWelcomeOnClick(message);
  };
  const fetchWelcomeOnLoad = async () => {
    const welcome = await client.fetchWelcomeMessage();
    setWelcomeOnLoad(welcome);
  };
  useEffect(() => {
    fetchWelcomeOnLoad();
  }, []);
  return (
    <div>
      <h3>HTTP Client</h3> <hr />
      <h4>Requesting on Click</h4>
      ...
      <hr />
      <h4>Requesting on Load</h4>
      Response from server: <b>{welcomeOnLoad}</b>
      <hr />
    </div>
  );
}
```

### Requesting on Load

Response from server: **Welcome to Lab 5**

## 3.5.5 Working with Remote Objects on a Server Asynchronously

Let's now revisit the APIs that worked with the **assignment** object in **WorkingWithObjects** and create an asynchronous version. Let's add client functions to **client.ts** to fetch the assignment object from the server and update its title as shown below.

src/Labs/Lab5/client.ts

```
import axios from "axios";
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
export const fetchWelcomeMessage = async () => {
  const response = await axios.get(`${REMOTE_SERVER}/lab5/welcome`);
  return response.data;
};
const ASSIGNMENT_API = `${REMOTE_SERVER}/lab5/assignment`;
export const fetchAssignment = async () => {
  const response = await axios.get(`${ASSIGNMENT_API}`);
  return response.data;
};
export const updateTitle = async (title: string) => {
  const response = await axios.get(`${ASSIGNMENT_API}/title/${title}`);
  return response.data;
};
```

Then, in a new **WorkingWithObjectsAsynchronously** component, create a UI that fetches the assignment on load and then allows you to edit its title. Import the component in **Lab5** and confirm that the assignment is displayed on load.

src/Labs/Lab5/WorkingWithObjectsAsynchronously.tsx

```
import React, { useEffect, useState } from "react";
import * as client from "./client";
export default function WorkingWithObjectsAsynchronously() {
  const [assignment, setAssignment] = useState<any>({});
  const fetchAssignment = async () => {
    const assignment = await client.fetchAssignment();
    setAssignment(assignment);
  };
  useEffect(() => {
    fetchAssignment();
  }, []);
  return (
    <div id="wd-asynchronous-objects">
      <h3>Working with Objects Asynchronously</h3>
      <h4>Assignment</h4>
      <input defaultValue={assignment.title} className="form-control mb-2"
        onChange={(e) => setAssignment({ ...assignment, title: e.target.value }) } />
      <textarea defaultValue={assignment.description} className="form-control mb-2"
        onChange={(e) => setAssignment({ ...assignment, description: e.target.value }) } />
      <input type="date" className="form-control mb-2" defaultValue={assignment.due}
        onChange={(e) => setAssignment({ ...assignment, due: e.target.value }) } />
      <div className="form-check form-switch">
        <input className="form-check-input" type="checkbox" id="wd-completed"
          defaultChecked={assignment.completed}
          onChange={(e) => setAssignment({ ...assignment, completed: e.target.checked }) } />
        <label className="form-check-label" htmlFor="wd-completed"> Completed </label>
      </div>
      <pre>{JSON.stringify(assignment, null, 2)}</pre>
      <hr />
    </div>
  );
}
```

## Assignment

NodeJS Assignment

Create a NodeJS server  
with ExpressJS

10/10/2021



☒ Completed

Now let's add a button to update the assignment's title. Change the assignment's title, refresh the screen and confirm that the title has changed.

src/Labs/Lab5/WorkingWithObjectsAsynchronously.tsx

```
export default function WorkingWithObjectsAsynchronously() {
  const [assignment, setAssignment] = useState<any>({});
  const fetchAssignment = async () => {
    const assignment = await client.fetchAssignment();
    setAssignment(assignment);
  };
  const updateTitle = async (title: string) => {
    const updatedAssignment = await client.updateTitle(title);
    setAssignment(updatedAssignment);
  };
  ...
  return (
    <div id="wd-asynchronous-objects">
      <h3>Working with Objects Asynchronously</h3>
      <h4>Assignment</h4>
      ...
      <button className="btn btn-primary me-2" onClick={() => updateTitle(assignment.title)} >
        Update Title
      </button>
      <pre>{JSON.stringify(assignment, null, 2)}</pre>
      <hr />
    </div>
  );
}
```

## 3.5.6 Working with Remote Arrays on a Server Asynchronously

Now let's do the same thing to the arrays. Let's use **axios** so that we can manipulate remote arrays on the server from the user interface and update a user interface to reflect the changes in the remote array. We'll implement several client functions in **client.ts** that use **axios** to communicate with the server and we'll use them from a new component that will render the remote array in the user interface.

src/Labs/Lab5/client.ts

```
const TODOS_API = `${REMOTE_SERVER}/lab5/todos`;
export const fetchTodos = async () => {
  const response = await axios.get(TODOS_API);
  return response.data;
};
```

The exercise below fetches the **todos** from the server and populate **todos** state variable which we can then render as a list of todos when the component loads. Confirm that the todos render when the component first loads.

src/Labs/Lab5/WorkingWithArraysAsynchronously.tsx

```
import React, { useState, useEffect } from "react";
import * as client from "../client";
export default function WorkingWithArraysAsynchronously() {
  const [todos, setTodos] = useState<any[]>([]);
  const fetchTodos = async () => {
    const todos = await client.fetchTodos();
    setTodos(todos);
  };
  useEffect(() => {
    fetchTodos();
  }, []);
  return (
    <div id="wd-asynchronous-arrays">
      <h3>Working with Arrays Asynchronously</h3>
      <h4>Todos</h4>
      <ul className="list-group">
        {todos.map((todo) => (
          <li key={todo.id} className="list-group-item">
            <input type="checkbox" className="form-check-input me-2"
              defaultChecked={todo.completed}/>
            <span style={{ textDecoration: todo.completed ? "line-through" : "none" }}>
              {todo.title}
            </span>
          </li>
        ))}
      </ul> <hr />
    </div>
  );
};
```

## Todos

☐ Task 1

☒ Task 2

☐ Task 3

☒ Task 4

### 3.5.7 Deleting Data from a Server Asynchronously

In **client.ts**, add a **removeTodo** client function that sends a **delete** request to the server. The server will respond with an array with the surviving todos.

src/Labs/Lab5/client.ts

```
export const removeTodo = async (todo: any) => {
  const response = await axios.get(`${TODOS_API}/${todo.id}/delete`);
  return response.data;
};
```

In the **WorkingWithArraysAsynchronously** component, add **remove** buttons to each of the todos that invoke a new **removeTodo** function that uses the client to send asynchronous delete request to the server and updates the **todos** state variable with the surviving todos. Use a **trashcan** icon to represent the **remove** button as shown below. Confirm that clicking on the new **remove** buttons actually removes the corresponding todo.

src/Labs/Lab5/WorkingWithArraysAsynchronously.tsx

```
export default function WorkingWithArraysAsynchronously() {
  const [todos, setTodos] = useState<any[]>([]);
  const fetchTodos = async () => { ... };
  const removeTodo = async (todo: any) => {
```







```

const updatedTodos = await client.removeTodo(todo);
setTodos(updatedTodos);
};

...
return (
  <div id="wd-asynchronous-arrays">
    <h3>Working with Arrays Asynchronously</h3>
    <h4>Todos</h4>
    <ul className="list-group">
      {todos.map((todo) => (
        <li key={todo.id} className="list-group-item">
          <FaTrash onClick={() => removeTodo(todo)}
            className="text-danger float-end mt-1" id="wd-remove-todo"/>
          ...
        </li>
      ))}
    </ul><hr />
  </div>
);

```

## Todos

<input type="checkbox"/>	Task 1	
<input checked="" type="checkbox"/>	Task 2	
<input type="checkbox"/>	Task 3	
<input checked="" type="checkbox"/>	Task 4	

### 3.5.8 Creating New Data in a Server Asynchronously

A previous exercise implemented server route to create new todo items. In the React's project **client.ts** implement a **createTodo** client function that requests creating a new todo item from the server as shown below.

src/Labs/Lab5/client.ts

```

export const createTodo = async () => {
  const response = await axios.get(`${TODOS_API}/create`);
  return response.data;
};

```

In the **WorkingWithArraysAsynchronously** component, add a **+ button icon** to invoke the **createTodo** client function and update the **todos** state variables with the **todos** from the server. Confirm that clicking the **+ button icon** actually creates a new todo.

src/Labs/Lab5/WorkingWithArraysAsynchronously.tsx

```

import { FaPlusCircle } from "react-icons/fa";
import * as client from "./client";
export default function WorkingWithArraysAsynchronously() {
  const [todos, setTodos] = useState<any[]>([]);
  const createTodo = async () => {
    const todos = await client.createTodo();
    setTodos(todos);
  };
  ...
  return (
    <div id="wd-asynchronous-arrays">
      <h3>Working with Arrays Asynchronously</h3>
      <h4> Todos
        <FaPlusCircle onClick={createTodo} className="text-success float-end fs-3"
          id="wd-create-todo" /> </h4>
      ...
    </div>
  );
};

```

### 3.6 Passing JSON Data to a Server in an HTTP Body

The exercises so far have sent data to the server as path and query parameters. This approach is limited to the maximum length of the URL string, and only string data types. Another concern is that data in the URL is sent over a network in clear text, so anyone snooping around between the client and server can see the data as plain text which is not a good option for exchanging sensitive information such as passwords and other personal data. A better approach is to encode the data as **JSON** in the **HTTP request body** which allows for arbitrarily large amounts of data as well as secure data encryption. To

enable the server to parse **JSON** data from the **request body**, add the following **app.use()** statement in **index.js**. Make sure that it's implemented right after the **CORS** configuration statement. Now **JSON** data coming from the client is available in the **request body** in the **request.body** property in the server routes.

**index.js**

```
import Lab5 from "../Lab5/index.js";
import cors from "cors";
const app = express();
app.use(cors());
app.use(express.json());
Lab5(app);
app.listen(4000);

// make sure this statement occurs AFTER setting up CORS
```

The hyperlinks and **axios.get()** in the exercises so far have sent data to the server using the **HTTP GET method** or **verb**. HTTP defines several other **HTTP methods** or **verbs** including:

- **GET** - for retrieving data, but we've been also misusing it for creating, modifying and deleting data on the server. We'll start using it properly only for retrieving data
- **POST** - for creating new data typically embedded in the HTTP body
- **PUT** - for modifying existing data where updates are typically embedded in the HTTP body
- **DELETE** - for removing existing data
- **OPTIONS** - for retrieving allowed operations. Used to figure out if CORS policy allows communication with the other methods such as GET, POST, PUT and DELETE

The **GET** method, as the name suggests, is meant for only getting data from the server. We've been misusing it to implement routes that also creates, updates, and deletes data on the server. We did this mostly for academic purposes since it's the easiest HTTP method to work with. From now on we'll use the proper HTTP method for the right purpose.

### 3.6.1 Posting Data to Servers with HTTP POST Requests

To illustrate using the HTTP POST method, let's re-implement the route that creates new todos as shown below. The **HTTP POST method** takes the role of the verb meaning **create**. Add the new **app.post** implementation highlighted in green below. Don't remove the old version highlighted in yellow so we don't break the other lab exercises. Note how the new implementation grabs the posted **JSON** data from **req.body** and uses it to define **newTodo**. Also note that this version does not respond with the entire **todos** array and instead only responds with the newly created todo object instance. This is more reasonable since arrays can potentially be large and it would be expensive to transfer such large data structures over a network, especially if the client UI already has most of this data already displayed.

**Lab5/WorkingWithArrays.js**

```
...
app.get("/lab5/todos/create", (req, res) => {
  const newTodo = { id: new Date().getTime(), title: "New Task", completed: false };
  todos.push(newTodo);
  res.json(todos);
});
app.post("/lab5/todos", (req, res) => {
  const newTodo = { ...req.body, id: new Date().getTime() };
  todos.push(newTodo);
  res.json(newTodo);
});
...
```

Back in the user interface, add a new **postTodo** client function in **client.ts** that posts new **todo** objects to the server as shown below. Note the second argument in the **axios.post()** method containing the new **todo** object instance sent to the server. The response this time contains the **todo** instance added to the **todos** array in the server instead of all the todos on the server.

src/Labs/Lab5/client.ts

```
export const createTodo = async () => {
  const response = await axios.get(`${TODOS_API}/create`);
  return response.data;
};
export const postTodo = async (todo: any) => {
  const response = await axios.post(`${TODOS_API}`, todo);
  return response.data;
};
```

In the **WorkingWithArraysAsynchronously** component, create a new **+ button icon** to invoke the new **postTodo** client function to send a new **todo** object to the server containing a default **title** and **completed** properties. Append the new **todo** object created on the server, to the local **todos** state variable to update the user interface with the new todo as shown below. Color the new **+ button icon** a different color so it's distinguishable from the **createTodo** button. Confirm that you can add new items to the array when you click on the new **+ button icon**.

src/Labs/Lab5/WorkingWithArraysAsynchronously.tsx

```
import { FaPlusCircle } from "react-icons/fa";
import * as client from "../client";
export default function WorkingWithArraysAsynchronously() {
  const [todos, setTodos] = useState<any[]>([]);
  const createTodo = async () => {
    const todos = await client.createTodo();
    setTodos(todos);
  };
  const postTodo = async () => {
    const newTodo = await client.postTodo({ title: "New Posted Todo", completed: false, });
    setTodos([...todos, newTodo]);
  };
  return (
    <div id="wd-asynchronous-arrays">
      <h3>Working with Arrays Asynchronously</h3>
      <h4>
        Todos
        <FaPlusCircle onClick={createTodo} className="text-success float-end fs-3" id="wd-create-todo" />
        <FaPlusCircle onClick={postTodo} className="text-primary float-end fs-3 me-3" id="wd-post-todo" />
      </h4>
    </div>
  );
};
```

Todos



<input type="checkbox"/>	Task 1	
<input checked="" type="checkbox"/>	Task 2	
<input type="checkbox"/>	Task 3	
<input checked="" type="checkbox"/>	Task 4	
<input type="checkbox"/>	New Task	
<input type="checkbox"/>	New Posted Todo	

## 3.6.2 Deleting Data from Servers with HTTP DELETE Requests

Now that we have **axios** we can implement a better version of the remove operation. The current **removeTodo** implementation uses the **HTTP GET** method to request the server to remove data. The **HTTP DELETE** method is specifically suited for removing data from remote servers. In the server project, implement a better version of the delete operation as shown below. The new implementation uses the **HTTP DELETE** method declared in **app.delete()** which is distinct from **app.get()** for which we don't need the trailing **/delete** at the end of the **URL**. Removing the element from the array is the same either way. Although we could again respond with the entire array of surviving todos, it is better to just respond with a success status and let the user interface update its state variable. This reduces unnecessary data communication between the client and server.

Lab5/WorkingWithArrays.js

```
...
app.get("/lab5/todos/:id/delete", (req, res) => {
  const { id } = req.params;
  const todo = todos.find((t) => t.id === parseInt(id));
  const todoIndex = todos.indexOf(todo);
  if (todoIndex !== -1) {
    todos.splice(todoIndex, 1);
  }
  res.json(todos);
});
```

```
});
app.delete("/lab5/todos/:id", (req, res) => {
  const { id } = req.params;
  const todoIndex = todos.findIndex((t) => t.id === parseInt(id));
  todos.splice(todoIndex, 1);
  res.sendStatus(200);
});
```

In the React.js project create a new client function called **deleteTodo** as shown below. Note how it's implemented using **axios.delete** instead of **axios.get** so that it matches the server's **app.delete** as well as the **URL** format without the trailing **/delete**.

src/Labs/Lab5/client.ts

```
export const removeTodo = async (todo: any) => {
  const response = await axios.get(`${TODO_API}/${todo.id}/delete`);
  return response.data;
};
export const deleteTodo = async (todo: any) => {
  const response = await axios.delete(`${TODO_API}/${todo.id}`);
  return response.data;
};
```













In the user interface component **WorkingWithArraysAsynchronously**, add another **remove** button to try this new **deleteTodo** client function, but use a different icon, say an **X** so as to not confuse it with the **trash**. Note that the new implementation ignores the response from the server and instead filters the removed todo from the local state variable. This is fine for now, but the operation is too optimistic assuming the server successfully deleted the item from the array and updating the user interface without confirmation. Later we'll deal with errors from the server to make sure the local state variable in the user interface is in synch with the remote array on the server.

src/Labs/Lab5/WorkingWithArraysAsynchronously.tsx

```
import { TiDelete } from "react-icons/ti";
import * as client from "./client";
export default function WorkingWithArraysAsynchronously() {
  const [todos, setTodos] = useState<any[]>([]);
  ...
  const deleteTodo = async (todo: any) => {
    await client.deleteTodo(todo);
    const newTodos = todos.filter((t) => t.id !== todo.id);
    setTodos(newTodos);
  };
  ...
  return (
    ...
    <li key={todo.id} className="list-group-item">
      <FaTrash onClick={() => removeTodo(todo)} className="text-danger float-end mt-1" id="wd-remove-todo" />
      <TiDelete onClick={() => deleteTodo(todo)} className="text-danger float-end me-2 fs-3" id="wd-delete-todo" />
    </li>
    ...
  );
};
```

Todos



<input type="checkbox"/> Task 1		
<input checked="" type="checkbox"/> Task 2		
<input type="checkbox"/> Task 3		
<input checked="" type="checkbox"/> Task 4		
<input type="checkbox"/> New Task		
<input type="checkbox"/> New Posted Todo		

### 3.6.3 Updating Data on Servers with HTTP PUT Requests

Use **HTTP PUT** to reimplement the route that updates an item in an array as shown below. The route replaces the todo item whose ID matches the **id** path parameter with a combination of the original todo object and properties in the **req.body**. This overrides any properties in the original todo object with matching properties in the **req.body**. Note that this new implementation does not respond with the **todos** array, but instead responds with a simple **OK** status code of **200**. This is more reasonable since there is no need to respond with an entire array since the user interface already has the array cached in the browser and it can just update the item in the **todos** state variable.

#### Lab5/WorkingWithArrays.js

```
app.put("/lab5/todos/:id", (req, res) => {
  const { id } = req.params;
  todos = todos.map((t) => {
    if (t.id === parseInt(id)) {
      return { ...t, ...req.body };
    }
    return t;
  });
  res.sendStatus(200);
});
```

Back in the user interface, in **clients.ts** add a new **updateTodo** function that **puts** updates to the server as shown below. Note the second argument in the **axios.put()** method containing the updated **todo** object instance sent to the server. The response contains a status.

#### src/Labs/Lab5/client.ts



```
export const updateTodo = async (todo: any) => {
  const response = await axios.put(`${TODO_API}/${todo.id}`, todo);
  return response.data;
};
```

In the **WorkingWithArraysAsynchronously** component, add an input field that shows up when you click a new **pencil icon** by setting the **todo's editing** property to true. Pressing the **Enter** key sets the **todo's editing** property to false, and shows the updated **title** again. Add an **onChange** attribute to the **completed** checkbox so that it updates the corresponding property of the **todo** object. Confirm you can edit the **title** and **completed** properties of the **todos**, and that the changes persist after refreshing the page.

#### src/Labs/Lab5/WorkingWithArraysAsynchronously.tsx




```
import { FaPencil } from "react-icons/fa6";
export default function WorkingWithArraysAsynchronously() {
  const [todos, setTodos] = useState<any[]>([]);
  const editTodo = (todo: any) => {
    const updatedTodos = todos.map(
      (t) => t.id === todo.id ? { ...todo, editing: true } : t
    );
    setTodos(updatedTodos);
  };
  const updateTodo = async (todo: any) => {
    await client.updateTodo(todo);
    setTodos(todos.map((t) => (t.id === todo.id ? todo : t)));
  };
  return (
    <div id="wd-asynchronous-arrays">
      <h3>Working with Arrays Asynchronously</h3>
      <ul className="list-group">
        {todos.map((todo) => (
          <li key={todo.id} className="list-group-item">
            <FaPencil onClick={() => editTodo(todo)} className="text-primary float-end me-2 mt-1" />
            <input type="checkbox" defaultChecked={todo.completed} className="form-check-input me-2 float-start"
              onChange={(e) => updateTodo({ ...todo, completed: e.target.checked })} />
            {!todo.editing ? (todo.title) : (
              <input className="form-control w-50 float-start" defaultValue={todo.title}
                onKeyDown={(e) => {
                  if (e.key === "Enter") {
                    updateTodo({ ...todo, editing: false });
                  }
                }}
              />
            )}
            <input type="text" value={todo.title} onChange={(e) => updateTodo({ ...todo, title: e.target.value })} />
          )}
        )}
      </ul>
    </div>
  );
}
```

#### Todos






☐

Task 123






☒

Task-2






☐

Task 3






☒

Task-4






☐

New Task



☐

New Posted Todo



## 3.6.4 Handling Errors

The exercises so far have been very optimistic when interacting with the server, but it is good practice to handle edge cases and the unforeseen. In this section we're going to add error handling to some of the routes and user interface. For instance, the exercise below throws exceptions if the items being deleted or updated don't actually exist. Errors are reported by the server as status codes, where 404 is the infamous NOT FOUND error. Additionally a JSON object can be sent back as part of the response that can be used by user interfaces to better inform the user of what went wrong.

### Lab5/WorkingWithArrays.js

```
app.delete("/lab5/todos/:id", (req, res) => {
  const { id } = req.params;
  const todoIndex = todos.findIndex((t) => t.id === parseInt(id));
  if (todoIndex === -1) {
    res.status(404).json({ message: `Unable to delete Todo with ID ${id}` });
    return;
  }
  todos.splice(todoIndex, 1);
  res.sendStatus(200);
});
app.put("/lab5/todos/:id", (req, res) => {
  const { id } = req.params;
  const todoIndex = todos.findIndex((t) => t.id === parseInt(id));
  if (todoIndex === -1) {
    res.status(404).json({ message: `Unable to update Todo with ID ${id}` });
    return;
  }
  todos = todos.map((t) => { ... });
  res.sendStatus(200);
});
```













In the user interface we can **catch** the errors by wrapping the request in a **try/catch** clause as shown below. If the request fails with a HTTP error response, then the body of the **try** block is aborted and the body of the **catch** clause executes instead. The exercise below declares a **errorMessage** state variable that we populate with the error from the server if an error occurs. The error is rendered as an alert box as shown here on the right. To test, remove an item using the <http://localhost:4000/lab5/todos/:id/delete> route and then try to update or delete the same item using the user interface. Confirm you get an error if you try to delete or update a **todo** that does not exist.

### src/Labs/Lab5/WorkingWithArrays.tsx

```
export default function WorkingWithArraysAsynchronously() {
  const [errorMessage, setErrorMessage] = useState(null);
  const updateTodo = async (todo: any) => {
    try {
      await client.updateTodo(todo);
      setTodos(todos.map((t) => (t.id === todo.id ? todo : t)));
    } catch (error: any) {
      setErrorMessage(error.response.data.message);
    }
  };
  const deleteTodo = async (todo: any) => {
    try {
      await client.deleteTodo(todo);
      const newTodos = todos.filter((t) => t.id !== todo.id);
      setTodos(newTodos);
    } catch (error: any) {
      console.log(error);
      setErrorMessage(error.response.data.message);
    }
  };
  ...
  return (
    <div id="wd-asynchronous-arrays">
      <h3>Working with Arrays Asynchronously</h3>
      {errorMessage && <div id="wd-todo-error-message" className="alert alert-danger mb-2 mt-2">{errorMessage}</div>}}
      ...
    </div>
  );
};
```

Unable to delete Todo with ID 123

### Todos

<input type="checkbox"/> Task 1			
<input checked="" type="checkbox"/> Task 2			
<input type="checkbox"/> Task 3			
<input checked="" type="checkbox"/> Task 4			

## 4 Implementing the Kanbas Node.js HTTP Server

Kanbas is currently implemented entirely as a React.js application. Although various CRUD operations have been implemented to create, read, update, and delete courses and modules, these changes are not permanent and are lost when the browser is refreshed. To make the changes permanent, it is necessary to integrate the React.js user interface with a server that can access resources such as the file system, network, and database. In this section, server routes will be implemented to integrate the user interface with the server. In the next chapter, changes will be stored permanently in a MongoDB non-relational database.

### 4.1 Migrating the "Database" to the Server

In previous assignments, a **Database** component was created to consolidate all data files into a single access point. Ideally, the data should reside on the server side or within a dedicated database. In this assignment, the data will first be moved to the server, with a transition to a database in the subsequent assignment. Begin by creating a folder named **Kanbas** at the root of the Node.js project. Inside the **Kanbas** folder, create a **Database** directory and copy all JSON files from the React.js project. Then, change the file extensions of the JSON files to JavaScript, for example, rename **users.json** to **users.js** and **courses.json** to **courses.js**. At the top of each newly converted JavaScript file, include an export default statement, as shown below.

*Kanbas/Database/courses.js*

```
export default [
  { _id: "RS101", name: "Rocket Propulsion", number: "RS4550", startDate: "2023-01-10",
    endDate: "2023-05-15", department: "D123", credits: 4, description: "..."}, ...
];
```

Do the same for all the JSON files and update the import statements in the **Database** component as shown below. Use the same data files from previous assignments. Feel free to modify the data in the files to customize the content or meet requirements in this assignment. Ignore unnecessary data files.

*Kanbas/Database/index.js*

```
import courses from "./courses.js";
import modules from "./modules.js";
import assignments from "./assignments.js";
import users from "./users.js";
import grades from "./grades.js";
import enrollments from "./enrollments.js";
export default { courses, modules, assignments, users, grades, enrollments };
```

### 4.2 Integrating the Account Screens with the Server with RESTful Web APIs

The **Data Access Object (DAO)** design pattern organizes data access by grouping it based on data types or collections. The following **Users/dao.js** file implements various CRUD operations for handling the **users** array in the **Database**. Later sections in the assignment will create additional **DAOs** for each of the data arrays: courses, modules, etc.

*Kanbas/Users/dao.js*

```
import db from "../Database/index.js";
let { users } = db;
export const createUser = (user) => {
  const newUser = { ...user, _id: Date.now() };
  users = [...users, newUser];
  return newUser;
};
export const findAllUsers = () => users;
export const findUserById = (userId) => users.find((user) => user._id === userId);
export const findUserByUsername = (username) => users.find((user) => user.username === username);
```



```
export const findUserByCredentials = (username, password) =>
  users.find( (user) => user.username === username && user.password === password );
export const updateUser = (userId, user) => (users = users.map((u) => (u._id === userId ? user : u)));
export const deleteUser = (userId) => (users = users.filter((u) => u._id !== userId));
```

## 4.2.1 Integrating the React Sign In Screen with a RESTful Web API

**DAOs** provide an interface between an application and low-level database access, offering a high-level API to the rest of the application while abstracting the details and idiosyncrasies of using a particular database vendor. Similarly, routes create an interface between the HTTP network layer and the JavaScript object and function layer by transforming a stream of bits from a network connection request into a set of objects, maps, and function event handlers that are part of the client/server architecture in a multi-tiered application.

The Node.js server implements uses routes to integrate with the user interface and implements DAOs to communicate with the **Database**. The server functions between these two layers, which is why it is often called the **middle tier** in a **multi-tiered** application. The following routes expose the database operations through a RESTful API, and the implementation of each function will be covered in the following sections. This assignment uses the **Database** component implemented in **Database/index.ts**. Later assignments will refactor this by using an actual database.

*Kanbas/Users/routes.js*

```
import * as dao from "../dao.js";
let currentUser = null;
export default function UserRoutes(app) {
  const createUser = (req, res) => { };
  const deleteUser = (req, res) => { };
  const findAllUsers = (req, res) => { };
  const findUserById = (req, res) => { };
  const updateUser = (req, res) => { };
  const signup = (req, res) => { };
  const signin = (req, res) => { };
  const signout = (req, res) => { };
  const profile = (req, res) => { };
  app.post("/api/users", createUser);
  app.get("/api/users", findAllUsers);
  app.get("/api/users/:userId", findUserById);
  app.put("/api/users/:userId", updateUser);
  app.delete("/api/users/:userId", deleteUser);
  app.post("/api/users/signup", signup);
  app.post("/api/users/signin", signin);
  app.post("/api/users/signout", signout);
  app.post("/api/users/profile", profile);
}
```

Import and configure the routes in **index.js** as shown below.

*index.js*

```
import express from "express";
...
import UserRoutes from "../Kanbas/Users/routes.js";

const app = express();
UserRoutes(app);
...
app.listen(process.env.PORT || 4000);
```

**Routes** implement **RESTful Web APIs** that clients can use to integrate with server functionality. The route implemented below extracts properties **username** and **password** from the request's body and pass them to the **findUserByCredentials** function implemented by the DAO. The resulting user is stored in the server variable **currentUser** to remember the logged in user. The user is then sent to the client in the response. Later sections will add error handling.

#### Kanbas/Users/routes.js

```
import * as dao from "../dao.js";
let currentUser = null;
export default function UserRoutes(app) {
  ...
  const signin = (req, res) => {
    const { username, password } = req.body;
    currentUser = dao.findUserByCredentials(username, password);
    res.json(currentUser);
  };
  app.post("/api/users/signin", signin);
  ...
}
```

In the React user interface, under **Kanbas/Account**, implement the **client** shown below to integrate with the user routes implemented in the server. The client function **signin** shown below posts a **credentials** object containing the **username** and **password** expected by the server. If the credentials are found, the response should contain the logged in user.

#### src/Kanbas/Account/client.ts

```
import axios from "axios";
export const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
export const USERS_API = `${REMOTE_SERVER}/api/users`;

export const signin = async (credentials: any) => {
  const response = await axios.post(`${USERS_API}/signin`, credentials);
  return response.data;
};
```

Implement a **Sign in** screen users can use to authenticate with the application. The following component declares state variable **credentials** to edit the **username** and **password**. Clicking the **Sign in** button posts the **credentials** to the server using the **client.signin** function. When the server responds successfully, the currently logged in user is stored in the user reducer and navigate to the **Profile** screen implemented in a later section.

#### src/Kanbas/Account/Signin.tsx

```
import * as client from "../client";
export default function Signin() {
  const [credentials, setCredentials] = useState<any>({});
  const dispatch = useDispatch();
  const navigate = useNavigate();
  const signin = async () => {
    const user = await client.signin(credentials);
    if (!user) return;
    dispatch(setCurrentUser(user));
    navigate("/Kanbas/Dashboard");
  };
  return ( ... );
}
```

## 4.2.2 Integrating the React Sign Up Screen with a RESTful Web API

The DAO implements functions **createUser** and **findUserByUsername** as shown below. The **createUser** DAO function accepts a user object from the user interface and then inserts the user into the **Database**. The **findUserByUsername** accepts a **username** from the user interface and finds the user with the matching **username**.

#### Users/dao.js

```
export const createUser = (user) => (users = [...users, { ...user, _id: Date.now() }]);
export const findUserByUsername = (username) => users.find((user) => user.username === username);
```

The DAO functions are used to implement the **sign up** operation for users to sign up to the application. The **signup** route expects a user object with at least the properties **username** and **password**. The DAO's **findUserByUsername** is called to check if a user with that username already exists. If such a user is found a 400 error status is returned along with an error message for display in the user interface. If the username is not already taken the user is inserted into the database and stored in the **currentUser** server variable. The response includes the newly created user. The **signup** route is mapped to the **api/users/signup** path.

#### Users/routes.js

```
import * as dao from "../dao.js";
let currentUser = null;
...
export default function UserRoutes(app) {
  ...
  const signup = (req, res) => {
    const user = dao.findUserByUsername(req.body.username);
    if (user) {
      res.status(400).json({
        message: "Username already in use" });
      return;
    }
    currentUser = dao.createUser(req.body);
    res.json(currentUser);
  };
  app.post("/api/users/signup", signup);
  ...
}
```

Meanwhile in the React user interface Web app, implement a **signup** client that posts the new user to the Web API as shown below.

#### src/Kanbas/Account/client.ts

```
...
export const signup = async (user: any) => {
  const response = await axios.post(`${USERS_API}/signup`, user);
  return response.data;
};
...
```

If not already done so, implement a **Sign up** screen component that users can use to type their username and password, and post the credentials to the server for signing up. If the sign up is successful, navigate to the **Profile** screen. In the **Account** component, update the **signup** route to display the new **Sign up** screen. In the **Sign in** screen, create a **Link** to navigate to the **Sign up** screen. Confirm that you can signup with a new username and password. Style the **Sign up** screen so it looks as shown below on the right. Confirm navigates to profile and shows new user. Confirm you can navigate

#### src/Kanbas/Account/Signup.tsx

```
import React, { useState } from "react";
import { Link, useNavigate } from "react-router-dom";
import * as client from "../client";
import { useDispatch } from "react-redux";
import { setCurrentUser } from "../reducer";
export default function Signup() {
  const [user, setUser] = useState<any>({});
  const navigate = useNavigate();
  const dispatch = useDispatch();
  const signup = async () => {
    const currentUser = await client.signup(user);
    dispatch(setCurrentUser(currentUser));
    navigate("/Kanbas/Account/Profile");
  };
  return (
    <div className="wd-signup-screen">
      <h1>Sign up</h1>
```

## Signup

[Signin](#)

```

<input value={user.username} onChange={(e) => setUser({ ...user, username: e.target.value })}
      className="wd-username form-control mb-2" placeholder="username" />
<input value={user.password} onChange={(e) => setUser({ ...user, password: e.target.value })} type="password"
      className="wd-password form-control mb-2" placeholder="password" />
<button onClick={signup} className="wd-signup-btn btn btn-primary mb-2 w-100"> Sign up </button><br />
<Link to="/Kanbas/Account/Signin" className="wd-signin-link">Sign in</Link>
</div>
);}

```

## 4.2.3 Integrating the React Profile Screen with a RESTful Web API

In the **User's DAO**, implement **updateUser** as shown below to update a single document by first identifying it by its primary key, and then updating the matching fields in the **user** parameter.

*Users/dao.js*

```
export const updateUser = (userId, user) => (users = users.map((u) => (u._id === userId ? user : u)));
```

In the **User's routes**, make the **DAO** function available as a RESTful Web API as shown below. Map a route that accepts a user's primary key as a path parameter, passes the ID and request body to the DAO function and responds with the status.

*Users/routes.js*

```

export default function UserRoutes(app) {
  ...
  const updateUser = (req, res) => {
    const userId = req.params.userId;
    const userUpdates = req.body;
    dao.updateUser(userId, userUpdates);
    currentUser = dao.findUserById(userId);
    res.json(currentUser);
  };
  ...
  app.put("/api/users/:userId", updateUser);
}

```

In the React client application, add client function **updateUser** to send user updates to the server to be saved to the database.

*src/Kanbas/Account/client.ts*

```

export const updateUser = async (user: any) => {
  const response = await axios.put(`${USERS_API}/${user._id}`, user);
  return response.data;
};

```

In the **Profile** screen implement the **updateProfile** event handler as shown below to update the profile on the server. Add an **Update** button that invokes the new handler. Confirm that the profile changed by logging out and then logging back in.

*src/Kanbas/Account/client.ts*

```

...
import * as client from "../client";
export default function Profile() {
  const [profile, setProfile] = useState<any>({});
  const dispatch = useDispatch();
  const navigate = useNavigate();
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  const updateProfile = async () => {
    const updatedProfile = await client.updateUser(profile);
    dispatch(setCurrentUser(updatedProfile));
  };
  ...
}

```

```

return (
  <div id="wd-profile-screen">
    <h3>Profile</h3>
    {profile && (
      ...
      <div>
        <button onClick={updateProfile} className="btn btn-primary w-100 mb-2"> Update </button>
        <button ...> Sign out </button>
      </div>
    )}
  </div>
);}

```

## 4.2.4 Retrieving the Profile from the Server

When a successful sign in occurs, the account information is stored in a server variable called **currentUser**. The variable retains the signed-in user information as long as the server is running. The **Sign in** screen copies currentUser from the server into the **currentUser** state variable in the reducer and then navigates to the Profile screen. If the browser reloads, the **currentUser** state variable is cleared and the user is logged out. To address this, the browser must check whether someone is already logged in from the server and, if so, update the copy in the reducer. Create a route on the server to provide access to **currentUser** as shown below.

*Users/routes.js*

```

let currentUser = null;
export default function UserRoutes(app) {
  ...
  const signin = async (, res) => { ... };
  const profile = async (req, res) => {
    res.json(currentUser);
  };
  ...
  app.post("/api/users/signin", signin);
  app.post("/api/users/profile", profile);
}

```

Then in the React.js Web app, implement a function to retrieve the **account** information from the server route implemented above as shown below.

*src/Kanbas/Account/client.ts*

```

import axios from "axios";
export const USERS_API = process.env.REACT_APP_REMOTE_SERVER;
export const signin = async (user) => { ... };
export const profile = async () => {
  const response = await axios.post(`${USERS_API}/profile`);
  return response.data;
};

```

Create a new **Session** component that fetches the **current user** from the server and stores it in the store so that the rest of the application can have access to the **current user**.

*src/Account/Session.tsx*

```

import * as client from "../client";
import { useEffect, useState } from "react";
import { setCurrentUser } from "../reducer";
import { useDispatch } from "react-redux";
export default function Session({ children }: { children: any }) {
  const [pending, setPending] = useState(true);
  const dispatch = useDispatch();
  const fetchProfile = async () => {
    try {

```

```

    const currentUser = await client.profile();
    dispatch(setCurrentUser(currentUser));
  } catch (err: any) {
    console.error(err);
  }
  setPending(false);
};
useEffect(() => {
  fetchProfile();
}, []);
if (!pending) {
  return children;
}
}
}

```

Wrap the Kanbas application with **Session** component so that it renders before all other components to check if anyone is signed in. Once it figures out either way, it'll store the result in the store and let the rest of the components render. Confirm that it works by signing in and then from the Profile screen, reload the browser. Make sure the user information still renders correctly.

*src/Kanbas/index.ts*

```

...
import Session from "../Account/Session";
export default function Kanbas() {
  ...
  return (
    <Session>
      <div id="wd-kanbas">
        ...
      </div>
    </Session>
  );
}

```

## 4.2.5 Integrating Signout with a RESTful Web API

Implement a route for users to signout that resets the **currentUser** to null in the server as shown below.

*Users/routes.js*

```

let currentUser = null;
function UserRoutes(app) {
  ...
  const signout = (req, res) => {
    currentUser = null;
    res.sendStatus(200);
  };
  app.post("/api/users/signout", signout);
  ...
}
export default UserRoutes;

```

In the React user interface Web application, add a client function that can post to the **signout** route.

*src/Kanbas/Account/client.ts*

```

export const signout = async () => {
  const response = await axios.post(`${USERS_API}/signout`);
  return response.data;
};

```

In the **Profile** screen refactor the **signout** function to invoke the **signout** client function and then navigates to the **Sign in** screen. Confirm that you can signout and navigate to the **Sign in** screen.

src/Kanbas/Account/Profile.tsx

```
...
const signout = async () => {
  await client.signout();
  dispatch(setCurrentUser(null));
  navigate("/Kanbas/Account/Signin");
};
...
<button onClick={signout} className="wd-signout-btn btn btn-danger w-100">
  Sign out
</button>
...
```

## 4.3 Supporting Multiple User Sessions

The user authentication implemented so far is simple but supports only one signed-in user at a time. Web applications typically support multiple users signed in simultaneously. This section describes how to add session handling to the Node.js server to allow multiple users to be signed in at the same time.

### 4.3.1 Installing and Configuring Server Sessions

First, it is necessary to narrow down who is allowed to authenticate. Configure **CORS** to support cookies and restrict network access to come only from the React application as shown below.

index.js

```
const app = express();
app.use(
  cors({
    credentials: true, // support cookies
    origin: process.env.NETLIFY_URL || "http://localhost:3000", // restrict cross origin resource
  }) // sharing to the react application
);
app.use(express.json());
const port = process.env.PORT || 4000;
```

In the Node.js project, install the **express-session** library as shown below.

```
$ npm install express-session
```

Then in the server implementation file, import and configure the session library as shown below. Make sure to configure sessions **after** configuring cors.

index.js

```
import session from "express-session"; // import new server session library
const app = express();
app.use(cors({ ... })); // configure cors first
const sessionOptions = { // configure server sessions after cors
  secret: "any string", // this is a default session configuration that works fine
  resave: false, // locally, but needs to be tweaked further to work in a
  saveUninitialized: false, // remote server such as AWS, Render, or Heroku. See later
};
app.use(
  session(sessionOptions)
);
```

Install the **dotenv** library to read configurations from environment variables on the server.



```
$ npm install dotenv
```

In a new **.env** file at the root of the project, declare the following environment variables.

**.env**

```
NODE_ENV=development
NETLIFY_URL=http://localhost:3000
REMOTE_SERVER=http://localhost:4000
SESSION_SECRET=super secret session phrase
MONGO_CONNECTION_STRING=mongodb://localhost:27017/kanbas
```

In **index.js**, import the **dotenv** library to determine whether the application is running in the development environment, and configure the session accordingly. Note: the following configuration has been tested on **Google's Chrome** and **Apple's Safari** browsers.

**index.js**

```
import "dotenv/config"; // import the new dotenv library
import session from "express-session"; // to read .env file
const app = express();
app.use(
  cors({
    credentials: true,
    origin: process.env.NETLIFY_URL || "http://localhost:3000", // use different front end URL in dev
  }) // and in production
);
const sessionOptions = { // default session options
  secret: process.env.SESSION_SECRET || "kanbas",
  resave: false,
  saveUninitialized: false,
};
if (process.env.NODE_ENV !== "development") { // in production
  sessionOptions.proxy = true; // turn on proxy support
  sessionOptions.cookie = { // configure cookies for remote server
    sameSite: "none",
    secure: true,
    domain: process.env.NODE_SERVER_DOMAIN,
  };
}
app.use(session(sessionOptions));
```

The **signup** route retrieves the **username** from the request body. If a user with that username already exists, an error is returned. Otherwise, create the new user and store it in the session's **currentUser** property to remember that this new user is now the currently logged-in user.

**Users/routes.js**

```
import * as dao from "../dao.js";
let currentUser = null;
...
export default function UserRoutes(app) {
  ...
  const signup = (req, res) => {
    const user = dao.findUserByUsername(req.body.username);
    if (user) {
      res.status(400).json({ message: "Username already taken" });
      return;
    }
    const currentUser = dao.createUser(req.body);
    req.session["currentUser"] = currentUser;
    res.json(currentUser);
  };
}
```

An existing user can identify themselves by providing credentials. The **signin** route below looks up the user by their credentials, stores it in **currentUser** session, and responds with the user if they exist. Otherwise responds with an error.

#### Users/routes.js

```
const signin = (req, res) => {
  const { username, password } = req.body;
  const currentUser = dao.findUserByCredentials(username, password);
  if (currentUser) {
    req.session["currentUser"] = currentUser;
    res.json(currentUser);
  } else {
    res.status(401).json({ message: "Unable to login. Try again later." });
  }
};
```

If a user has already signed in, the **currentUser** can be retrieved from the session by using the **profile** route as shown below. If there is no **currentUser**, an error is returned.

#### Users/routes.js

```
const profile = (req, res) => {
  const currentUser = req.session["currentUser"];
  if (!currentUser) {
    res.sendStatus(401);
    return;
  }
  res.json(currentUser);
};
```

Users can be signed out by destroying the session.

#### Users/routes.js

```
const signout = (req, res) => {
  req.session.destroy();
  res.sendStatus(200);
};
```

If a user updates their profile, then the session must be kept in synch.

#### Users/routes.js

```
const updateUser = (req, res) => {
  const userId = req.params.userId;
  const userUpdates = req.body;
  dao.updateUser(userId, userUpdates);
  const currentUser = dao.findUserById(userId);
  req.session["currentUser"] = currentUser;
  res.json(currentUser);
};
```

## 4.3.2 Configuring Axios to Support Server Sessions

By default **axios** does not support cookies. To configure **axios** to include cookies in requests, use the **axios.create()** to create an instance of the library that includes cookies for credentials as shown below. Then replace all occurrences of the **axios** library with this new version **axiosWithCredentials**.

#### src/Kanbas/Account/client.ts

```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
export const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
export const USERS_API = `${REMOTE_SERVER}/api/users`;
export const signin = async (credentials: any) => {
  const response = await axiosWithCredentials.post(`${USERS_API}/signin`, credentials);
};
```

```

    return response.data;
  };
  export const profile = async () => {
    const response = await axiosWithCredentials.post(`${USERS_API}/profile`);
    return response.data;
  };
  export const signup = async (user: any) => {
    const response = await axiosWithCredentials.post(`${USERS_API}/signup`, user);
    return response.data;
  };
  export const signout = async () => {
    const response = await axiosWithCredentials.post(`${USERS_API}/signout`);
    return response.data;
  };
  export const updateUser = async (user: any) => {
    const response = await axiosWithCredentials.put(`${USERS_API}/${user._id}`, user);
    return response.data;
  };
};

```

## 4.4 Creating a RESTful Web API for Courses

Previous assignments implemented CRUD operations to create, read, update and delete courses in the Kanbas Dashboard. These changes were transient and were lost when users refreshed the browser. This section demonstrates implementing a RESTful Web API to integrate the **Dashboard** and **Courses** screen with the server. The API will migrate the CRUD operations from the user interface to the server where they belong.

### 4.4.1 Retrieving Courses

Now that the **Database** has been moved to the server, it must be made available to the React.js client application through a Web **API (Application Programming Interface)**. The exercises below make the courses accessible at <http://localhost:4000/api/courses> for the React.js user interface to integrate. First implement a DAO to retrieve all courses from the **Database**.

*Kanbas/Courses/dao.js*

```

import Database from "../Database/index.js";
export function findAllCourses() {
  return Database.courses;
}

```

Use the **DAO** to implement a route that retrieves all the courses.

*Kanbas/Courses/routes.js*

```

import * as dao from "../dao.js";
export default function CourseRoutes(app) {
  app.get("/api/courses", (req, res) => {
    const courses = dao.findAllCourses();
    res.send(courses);
  });
}

```

In **index.js** import the new routes and pass a reference to the express module to the routes. Make sure to work **AFTER** the **cors**, **session**, and **json use** statements. Point your browser to <http://localhost:4000/api/courses> and confirm the server responds with an array of courses.

*index.js*

```

import express from "express";
import Lab5 from "../Lab5/index.js";

```

```
import UserRoutes from "../Kanbas/Users/routes.js";
import CourseRoutes from "../Kanbas/Courses/routes.js";
import cors from "cors";
const app = express();
app.use(cors(...));
app.use(session(...));
app.use(express.json()); // do all your work after this line
UserRoutes(app);
CourseRoutes(app);
Lab5(app);
app.listen(4000);
```

Since the Dashboard displays courses a user is enrolled in, implement **findCoursesForEnrolledUser** as shown below to retrieve courses the current user is enrolled in.

#### Kanbas/Courses/dao.js

```
...
export function findCoursesForEnrolledUser(userId) {
  const { courses, enrollments } = Database;
  const enrolledCourses = courses.filter((course) =>
    enrollments.some((enrollment) => enrollment.user === userId && enrollment.course === course._id));
  return enrolledCourses;
}
```

Since enrolled courses are retrieved within the context of the currently logged in user, implement the following route to retrieve the courses in the user routes.

#### Kanbas/Users/routes.js

```
import * as dao from "../dao.js";
import * as courseDao from "../Courses/dao.js";
export default function UserRoutes(app) {
  ...
  const findCoursesForEnrolledUser = (req, res) => {
    let { userId } = req.params;
    if (userId === "current") {
      const currentUser = req.session["currentUser"];
      if (!currentUser) {
        res.sendStatus(401);
        return;
      }
      userId = currentUser._id;
    }
    const courses = courseDao.findCoursesForEnrolledUser(userId);
    res.json(courses);
  };
  app.get("/api/users/:userId/courses", findCoursesForEnrolledUser);
  ...
}
```

Back in the user interface, create a **client.ts** file under the **Kanbas/Courses** that implements all the course related communication between the user interface and the server. Start by implementing the **fetchAllCourses** client function as shown below.

#### src/Kanbas/Courses/client.ts

```
import axios from "axios";
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
const COURSES_API = `${REMOTE_SERVER}/api/courses`;
export const fetchAllCourses = async () => {
  const { data } = await axios.get(COURSES_API);
  return data;
};
```

But the Dashboard only displays the course the current logged in user is enrolled in, so in the **Users**'s client file, implement **findMyCourses** that retrieves the current user's courses using the new **findCoursesForEnrolledUser** end point.

*src/Kanbas/Account/client.ts*

```
import axios from "axios";
export const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
export const USERS_API = `${REMOTE_SERVER}/api/users`;
const axiosWithCredentials = axios.create({ withCredentials: true });
export const findMyCourses = async () => {
  const { data } = await axiosWithCredentials.get(`${USERS_API}/current/courses`);
  return data;
};
...
```

In the **Kanbas** component use **useEffect** to fetch the courses from the server on component load and update the **courses** state variable that populates the **Dashboard**. Use the **currentUser** in the **accountReducer** as a dependency so that if a different user logs in, the courses will be reloaded from the server. Remove **Database** references from the user interface since we don't need it anymore. Also initialize the **courses** state variable as empty since we won't have the database anymore.

*src/Kanbas/index.tsx*

```
import * as db from "../Database";
import * as client from "../Courses/client";
import * as userClient from "../Account/client";
export default function Kanbas() {
  const [courses, setCourses] = useState<any[]>([]);
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  const fetchCourses = async () => {
    let courses = [];
    try {
      courses = await userClient.findMyCourses();
    } catch (error) {
      console.error(error);
    }
    setCourses(courses);
  };
  useEffect(() => {
    fetchCourses();
  }, [currentUser]);
  ...
}
```

In the **Dashboard** screen, remove the filtering of courses by enrollments since the server is already doing that. Restart the server and user interface to confirm that the **Dashboard** renders the courses the current user is enrolled in. Sign in as different users and confirm only the courses the user is enrolled in display in the **Dashboard**.

*src/Kanbas/Dashboard.tsx*

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

## 4.4.2 Creating New Courses

Implement a route that creates a new course and adds it to the **Database**. The new course is passed in the HTTP body from the client and is appended to the end of the courses array in the **Database**. The new course is given a new unique identifier and sent back to the client in the response.

*Kanbas/Courses/dao.js*

```
...
export function createCourse(course) {
  const newCourse = { ...course, _id: Date.now().toString() };
  Database.courses = [...Database.courses, newCourse];
  return newCourse;
}
```

When a course is created, it needs to be associated with the creator. In a new **Enrollments/dao.js** file, implement **enrollUserInCourse** to enroll, or associate, a user to a course.

*Kanbas/Enrollments/dao.js*

```
import Database from "../Database/index.js";
export function enrollUserInCourse(userId, courseId) {
  const { enrollments } = Database;
  enrollments.push({ _id: Date.now(), user: userId, course: courseId });
}
```

In the **Users**'s routes, implement **createCourse** as shown below to create a new course and then enroll the **currentUser** in the new course. Respond with the **newCourse** so it can be rendered in the user interface.

*Kanbas/Users/routes.js*

```
import * as dao from "../dao.js";
import * as courseDao from "../Courses/dao.js";
import * as enrollmentsDao from "../Enrollments/dao.js";
export default function UserRoutes(app) {
  const createCourse = (req, res) => {
    const currentUser = req.session["currentUser"];
    const newCourse = courseDao.createCourse(req.body);
    enrollmentsDao.enrollUserInCourse(currentUser._id, newCourse._id);
    res.json(newCourse);
  };
  app.post("/api/users/current/courses", createCourse);
  ...
}
```

In the course's **client.ts**, add a **createCourse** client function that **posts** a new course to the server and returns the response's data which should be the brand new course created in the server.

*src/Kanbas/Account/client.ts*

```
import axios from "axios";
export const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
export const USERS_API = `${REMOTE_SERVER}/api/users`;
const axiosWithCredentials = axios.create({ withCredentials: true });
export const createCourse = async (course: any) => {
  const { data } = await axiosWithCredentials.post(`${USERS_API}/current/courses`, course);
  return data;
};
...
```

In the **Kanbas** component, refactor the **addCourse** function so that it **posts** the new course to the server and the new course in the response is appended to the end of the **courses** state variable. Confirm that creating a new course updates the user interface with the added course.

*src/Kanbas/index.tsx*

```
...
import * as userClient from "../Account/client";
export default function Kanbas() {
  const addNewCourse = async () => {
    const newCourse = await userClient.createCourse(course);
    setCourses([ ...courses, newCourse ]);
  };
  const findAllCourses = async () => {...};
  useEffect(() => {...}, []);
  return ( ... );
}
```

### 4.4.3 Deleting a Course

Implement a route that removes a course and all enrollments associated with the course. First implement a **deleteCourse** DAO function that filters the course by its ID and then filters out all enrollments by the course's ID as shown below.

*Kanbas/Courses/dao.js*

```
import Database from "../Database/index.js";
export function deleteCourse(courseId) {
  const { courses, enrollments } = Database;
  Database.courses = courses.filter((course) => course._id !== courseId);
  Database.enrollments = enrollments.filter(
    (enrollment) => enrollment.course !== courseId
  );
}
```

In the Course's routes, implement a **delete** route that parses the course's ID from the URL and uses the **deleteCourse** DAO function as shown below. If the deletion was successful, respond with status 204, meaning the operation was successful and there are no other details.

*Kanbas/Courses/routes.js*

```
import * as dao from "../dao.js";
export default function CourseRoutes(app) {
  app.delete("/api/courses/:courseId", (req, res) => {
    const { courseId } = req.params;
    dao.deleteCourse(courseId);
    res.sendStatus(204);
  });
  ...
}
```

In the course's **client.ts**, add a **deleteCourse** client function that **deletes** an existing course from the server and returns the status response from the server.

*src/Kanbas/Courses/client.ts*

```
export const deleteCourse = async (id: string) => {
  const { data } = await axios.delete(`${COURSES_API}/${id}`);
  return data;
};
```

In the **Kanbas** component, refactor the **deleteCourse** function so that it uses the **courseClient** to **delete** the course from the server and then filters out the course from the local **courses** state variable. Confirm that clicking the **Delete** button of the course actually removes from the **Dashboard**.

*src/Kanbas/index.tsx*

```
...
import * as userClient from "../Account/client";
import * as courseClient from "../Courses/client";
export default function Kanbas() {
  const [courses, setCourses] = useState<any[]>([]);
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  const deleteCourse = async (courseId: string) => {
    const status = await courseClient.deleteCourse(courseId);
    setCourses(courses.filter((course) => course._id !== courseId));
  };
  ...
}
```

## 4.4.4 Updating a Course

In the Course's DAO, implement **updateCourse** function to update a course in the **Database**. First lookup the course by its ID and then apply the updates to the course as shown below.

*Kanbas/Courses/dao.js*

```
...
export function updateCourse(courseId, courseUpdates) {
  const { courses } = Database;
  const course = courses.find((course) => course._id === courseId);
  Object.assign(course, courseUpdates);
  return course;
}
```

In the Course's routes, implement a **put** route that parses the **id** of course as a path parameter and updates uses the **updateCourse** DAO function to update the corresponding course with the updates in HTTP request body. If the update is successful, respond with status 204.

*Kanbas/Courses/routes.js*

```
import * as dao from "../dao.js";
export default function CourseRoutes(app) {
  app.put("/api/courses/:courseId", (req, res) => {
    const { courseId } = req.params;
    const courseUpdates = req.body;
    dao.updateCourse(courseId, courseUpdates);
    res.sendStatus(204);
  });
  ...
}
```

In the course's **client.ts**, add a **updateCourse** client function that **updates** an existing course in the server and returns the status response from the server.

*src/Kanbas/Courses/client.ts*

```
...
export const updateCourse = async (course: any) => {
  const { data } = await axios.put(`${COURSES_API}/${course._id}`, course);
  return data;
};
...
```



In the **Kanbas** component, refactor the **updateCourse** function so that it **puts** the updated course to the server and then swaps out the old corresponding course with the new version in the **course** state variable. Confirm that clicking the **Update** button actually updates the course in the **Dashboard**.

src/Kanbas/index.tsx

```
...
import * as courseClient from "../Courses/client";
export default function Kanbas() {
  const updateCourse = async () => {
    await courseClient.updateCourse(course);
    setCourses(courses.map((c) => {
      if (c._id === course._id) { return course; }
      else { return c; }
    }));
  };
  const addCourse = async () => {...};
  useEffect(() => {...}, []);
  return ( ... );
}
```

## 4.5 Creating a RESTful Web API for Modules

Now let's do the same thing we did for the courses, but for the modules. We'll need routes that deal with the modules similar to the operations we implemented for the courses. We'll need to implement all the basic **CRUD** operations: **create** modules, **read**/retrieve modules, **update** modules and **delete** modules. The main difference will be that modules exist with the context of a particular course. Each course has a different set of modules, so the routes will need to take into account the course ID for which the modules we are operating on.

### 4.5.1 Retrieving a Course's Modules

Create **DAO** for the **Modules** to implement module data access from the **Database**. Start by implementing **findModulesForCourse** to retrieve a course's modules by its ID as shown below.

Kanbas/Modules/dao.js

```
import Database from "../Database/index.js";
export function findModulesForCourse(courseId) {
  const { modules } = Database;
  return modules.filter((module) => module.course === courseId);
}
```

In the routes file for the **Course** add a route to retrieve the modules for a course by its ID encoded in the path. Parse the course ID from the path and then use the module's DAO **findModulesForCourse** function to retrieve the modules for that course.

Kanbas/Courses/routes.js

```
import * as dao from "../dao.js";
import * as modulesDao from "../Modules/dao.js";
export default function CourseRoutes(app) {
  app.get("/api/courses/:courseId/modules", (req, res) => {
    const { courseId } = req.params;
    const modules = modulesDao.findModulesForCourse(courseId);
    res.json(modules);
  });
  ...
}
```

In the Course's **client.js** file in the React.js project, create **findModulesForCourse** to integrate the user interface with the server as shown below. Implement **findModulesForCourse** function shown below which retrieves the modules for a given course.

*src/Kanbas/Courses/client.tsx*

```
import axios from "axios";
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
const COURSES_API = `${REMOTE_SERVER}/api/courses`;
export const findModulesForCourse = async (courseId: string) => {
  const response = await axios
    .get(`${COURSES_API}/${courseId}/modules`);
  return response.data;
};
...
```

Update the modules reducer implemented in earlier assignments as shown below. Remove dependencies from the **Database** since we've moved it to the server. Empty the **modules** state variable since we'll be populating it with the modules we retrieve from the server using the **findModulesForCourse** function. Add a **setModules** reducer function so we can populate the **modules** state variable when we retrieve the modules from the server.

*src/Kanbas/Courses/Modules/reducer.tsx*

```
import db from "../Database";
const initialState = {
  modules: [],
};
const modulesSlice = createSlice({
  name: "modules",
  initialState,
  reducers: {
    setModules: (state, action) => {
      state.modules = action.payload;
    },
    addModule: (...) => {...},
    deleteModule: (...) => {...},
    updateModule: (...) => {...},
    editModule: (...) => {...},
  },
});
export const { addModule, deleteModule, updateModule, editModule, setModules } = modulesSlice.actions;
export default modulesSlice.reducer;
```

In the **Modules** component, import the new **setModules** reducer function and the new **findModulesForCourse** client function. Using a **useEffect** function, invoke the **findModulesForCourse** client function and dispatch the modules from the server to the reducer with the **setModules** function as shown below. Remove the filter since modules are already filtered on the server. Confirm that navigating to a course populates the corresponding modules.

*src/Kanbas/Courses/Modules/index.tsx*

```
import { setModules, addModule, editModule, updateModule, deleteModule } from "../reducer";
import { useState, useEffect } from "react";
import * as coursesClient from "../client";
export default function Modules() {
  const { cid } = useParams();
  const dispatch = useDispatch();
  const fetchModules = async () => {
    const modules = await coursesClient.findModulesForCourse(cid as string);
    dispatch(setModules(modules));
  };
  useEffect(() => {
    fetchModules();
  }, []);

  return (
    <div>
```

```

    ...
    <ul id="wd-modules" className="list-group rounded-0">
      {modules
        .filter((module: any) => module.course === cid)
        .map((module: any) => ( ... ))}
    </ul>
  </div>
);
}

```

## 4.5.2 Creating Modules for a Course

To create a new module in the **Database**, implement **createModule** in the Module's DAO as shown below. The function accepts the new module as a parameter, set its primary key and then appends the new module to the **Database's** module array.

*Kanbas/Modules/dao.js*

```

import Database from "../Database/index.js";
export function createModule(module) {
  const newModule = { ...module, _id: Date.now().toString() };
  Database.modules = [...Database.modules, newModule];
  return newModule;
}
export function findModulesForCourse(courseId) { ... }

```

In the **Course's** routes, implement a **post** Web API for the user interface to integrate to when creating a new module for a given course. Parse the course's ID from the path and the new module from the request's body. Set the new module's course's property to the course's ID so that the module know what course it belongs to. Use the module's DAO's createModule function to create the new module and then respond with the new module.

*Kanbas/Courses/routes.js*

```

import Database from "../Database/index.js";
import * as dao from "../dao.js";
import * as modulesDao from "../Modules/dao.js";
export default function CourseRoutes(app) {
  app.post("/api/courses/:courseId/modules", (req, res) => {
    const { courseId } = req.params;
    const module = {
      ...req.body,
      course: courseId,
    };
    const newModule = modulesDao.createModule(module);
    res.send(newModule);
  });
  app.get("/api/courses/:courseId/modules", (req, res) => { ... });
  ...
}

```

In the user interface, implement a **createModuleForCourse** client function that posts a new module from the user interface to the server as shown below. Encode the course's ID in the URL so the server can know what course the module belongs to.

*src/Kanbas/Courses/client.ts*

```

import axios from "axios";
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
const COURSES_API = `${REMOTE_SERVER}/api/courses`;
export const createModuleForCourse = async (courseId: string, module: any) => {
  const response = await axios.post(
    `${COURSES_API}/${courseId}/modules`,
    module
  );
};

```

```

    return response.data;
  };
  export const findModulesForCourse = async (courseId: string) => { ... };
  ...

```

In the **Modules** screen, implement a **createModuleForCourse** event handler that uses the new **createModuleForCourse** client function to send the module to the server and then dispatches the created module to the reducer so it's added to the reducer's **modules** state variable. Update the **ModulesControls addModule** attribute so that it uses the new **createModuleForCourse** event handler. Confirm that new modules are created for the current course.

*src/Kanbas/Courses/Modules/index.ts*

```

import { addModule, editModule, updateModule, deleteModule, setModules } from "../reducer";
import { useSelector, useDispatch } from "react-redux";
import * as coursesClient from "../client";
export default function Modules() {
  const { cid } = useParams();
  const [moduleName, setModuleName] = useState("");
  const { modules } = useSelector((state: any) => state.modulesReducer);
  const dispatch = useDispatch();
  const createModuleForCourse = async () => {
    if (!cid) return;
    const newModule = { name: moduleName, course: cid };
    const module = await coursesClient.createModuleForCourse(cid, newModule);
    dispatch(addModule(module));
  };

  return (
    <div>
      <ModulesControls setModuleName={setModuleName} moduleName={moduleName} addModule={createModuleForCourse} />
      ...
    </div>
  );
}

```

## 4.5.3 Deleting a Module

In the **Modules DAO**, implement **deleteModule** to remove a module from the **Database** by its **ID** as shown below.

*Kanbas/Modules/dao.js*

```

import Database from "../Database/index.js";
export function deleteModule(moduleId) {
  const { modules } = Database;
  Database.modules = modules.filter((module) => module._id !== moduleId);
}
export function createModule(module) { ... }
export function findModulesForCourse(courseId) { ... }

```

Create a **router** file for the **Modules** and implement a route that handles an **HTTP DELETE** to remove a module by its ID. Parse the module's ID from the path and use the **DAO's deleteModule** function to remove the module from the **Database**. If successful, respond with status 204.

*Kanbas/Modules/routes.js*

```

import * as modulesDao from "../dao.js";
export default function ModuleRoutes(app) {
  app.delete("/api/modules/:moduleId", (req, res) => {
    const { moduleId } = req.params;
    modulesDao.deleteModule(moduleId);
    res.sendStatus(204);
  });
}

```

In the **index.js** server file, import the new **ModuleRoutes** and pass it a reference to the **express** library.

*index.js*

```
...
import ModuleRoutes from "../Kanbas/Modules/routes.js";
...
app.use(express.json());
Lab5(app);
UserRoutes(app);
CourseRoutes(app);
EnrollmentRoutes(app);
ModuleRoutes(app);
app.listen(4000);
```

In the React.js user interface, create a new **client** file for **Modules**, and implement the **deleteModule** function as shown below. Pass it the **ID** of the module to be removed, encode it in a URL, and send it as an **HTTP DELETE** to the server.

*src/Kanbas/Courses/Modules/client.ts*

```
import axios from "axios";
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
const MODULES_API = `${REMOTE_SERVER}/api/modules`;
export const deleteModule = async (moduleId: string) => {
  const response = await axios.delete(`${MODULES_API}/${moduleId}`);
  return response.data;
};
```

In the **Modules** screen, import the new **client** file and use it to create the **removeModule** event handler as shown below. Use the new **deleteModule** client function to remove the module from the server. If successful, dispatch the deleted module's ID to the reducer to remove the module from the **modules** state variable as well. In the **ModuleControlsButtons** component, update the **deleteModule** attribute to use the new **removeModule** event handler. Confirm that clicking the trashcan of modules removes the modules. Refresh the screen to make sure that the modules is permanently deleted.

*src/Kanbas/Courses/Modules/index.js*

```
import { addModule, editModule, updateModule, deleteModule, setModules } from "../reducer";
import { useSelector, useDispatch } from "react-redux";
import * as coursesClient from "../client";
import * as modulesClient from "../client";
export default function Modules() {
  const { cid } = useParams();
  const [moduleName, setModuleName] = useState("");
  const { modules } = useSelector((state: any) => state.modulesReducer);
  const dispatch = useDispatch();
  const removeModule = async (moduleId: string) => {
    await modulesClient.deleteModule(moduleId);
    dispatch(deleteModule(moduleId));
  };
  ...
  return (
    <div>
      ...
      <ul id="wd-modules" className="list-group rounded-0">
        {modules.map((module: any) => (
          ...
          <ModuleControlButtons moduleId={module._id}
            deleteModule={(moduleId) => removeModule(moduleId)}
            editModule={(moduleId) => dispatch(editModule(moduleId))} />
        ))}
      </ul>
    </div>
  );
};
```

## 4.5.4 Update Module

In the **Module's DAO**, implement **updateModule** to update a module in the **Database** by its **ID**. First lookup the module by its **ID** and then apply the updates to the module as shown below.

#### Kanbas/Modules/dao.js

```
import Database from "../Database/index.js";
export function updateModule(moduleId, moduleUpdates) {
  const { modules } = Database;
  const module = modules.find((module) => module._id === moduleId);
  Object.assign(module, moduleUpdates);
  return module;
}
export function deleteModule(moduleId) { ... }
export function createModule(module) { ... }
export function findModulesForCourse(courseId) { ... }
```

In the **Module's routes** file, implement an **HTTP PUT** request handler that parses the **ID** of the course from the **URL** and the module updates from the **HTTP** request body. Use the **DAO's updateModule** function to apply the updates to the module. If successful, respond with **HTTP** status 204.

#### Kanbas/Modules/routes.js

```
import * as modulesDao from "../dao.js";
export default function ModuleRoutes(app) {
  app.put("/api/modules/:moduleId", (req, res) => {
    const { moduleId } = req.params;
    const moduleUpdates = req.body;
    modulesDao.updateModule(moduleId, moduleUpdates);
    res.sendStatus(204);
  });
  app.delete("/api/modules/:moduleId", (req, res) => { ... });
}
```

In the **client** file for **Modules** in the React.js user interface, implement the **updateModule** function as shown below. Pass it the module to be update. Encode the **ID** of the module in a URL, and sent the module updates in the body of an **HTTP PUT** request to the server.

#### src/Kanbas/Courses/Modules/client.ts

```
import axios from "axios";
const REMOTE_SERVER = process.env.REACT_APP_REMOTE_SERVER;
const MODULES_API = `${REMOTE_SERVER}/api/modules`;
export const updateModule = async (module: any) => {
  const { data } = await axios.put(`${MODULES_API}/${module._id}`, module);
  return data;
};
export const deleteModule = async (moduleId: string) => { ... };
```

In the **Modules** screen, implement a **saveModule** event handler as shown below. Use the new **updateModule** client function to update the module in the server. If successful, dispatch the updated module to the reducer to update the module in the **modules** state variable as well. In the **onKeyDown** event handler, invoke the **saveModule** event handler. Confirm that updating the module in the user interface actually modifies the module on the server. Refresh the screen to make sure that the modules has been modified.

#### src/Kanbas/Courses/Modules/index.tsx

```
...
import { addModule, editModule, updateModule, deleteModule, setModules } from "../reducer";
import { useSelector, useDispatch } from "react-redux";
import * as coursesClient from "../client";
import * as modulesClient from "../client";
export default function Modules() {
  const { cid } = useParams();
  const [moduleName, setModuleName] = useState("");
  const { modules } = useSelector((state: any) => state.modulesReducer);
  const dispatch = useDispatch();
  const saveModule = async (module: any) => {
```

```

    await modulesClient.updateModule(module);
    dispatch(updateModule(module));
  };
  ...
  return (
    <div>
      ...
      <ul id="wd-modules" className="list-group rounded-0">
        {modules.map((module: any) => (
          ...
          {!module.editing && module.name}
          { module.editing && (
            <input className="form-control w-50 d-inline-block" value={module.name}/>
            onChange={(e) => dispatch(updateModule({ ...module, name: e.target.value })) }
            onKeyDown={(e) => {
              if (e.key === "Enter") {
                saveModule({ ...module, editing: false });
              }
            }}
          )}
        )}
      </ul>
    </div>
  );
};

```

## 4.6 Assignments and Assignments Editor (On Your Own)

In your Node.js server application, implement routes for **creating**, **retrieving**, **updating**, and **deleting** assignments. In the React.js Web application, create an **assignment client** file that uses **axios** to send **POST**, **GET**, **PUT**, and **DELETE HTTP** requests to integrate the React.js application with the server application. In the React.js user interface, refactor the **Assignments** and **AssignmentEditor** screens implemented in earlier assignments to use the new **client** file to **CRUD** assignments. New assignments, updates to assignments, and deleted assignments should persist if the screens are refreshed as long as the server is running.

## 4.7 Enrollments (On Your Own)

In your Node.js server application, implement routes to support the Enrollments screen. Users should be able to enroll and unenroll from courses. In the React.js application, implement an enrollments client that uses **axios** to integrate with the routes in the server. Enrollments should persist as long as the server is running.

## 4.8 People Table (Optional)

In your Node.js server application, implement routes to support the People screen. Users should be able to see all users enrolled in the course. Faculty should be able to create, update, and delete users. In the React.js application, implement an users client that uses **axios** to integrate with the routes in the server. User changes should persist as long as the server is running.

# 5 Deploying RESTful Web Service APIs to a Public Remote Server

Up to this point you should have a working two tiered application with the first tier consisting of a front end React user interface application and the second tier consisting of a Node Express HTTP server application. In this section we're going to learn how to replicate this setup so that it can execute on remote servers. All development should be done in the local development environment on your personal development computer, and only when we're satisfied that all works fine locally, then we can make an effort at deploying the application on remote servers. The React Web application is already configured to deploy and run remotely on Netlify when you commit and push to the GitHub repository containing the source for the project. This section demonstrates how to configure the Node Express HTTP server project to deploy to a remote server hosted by **Render** or **Heroku** and then integrate the remote React Web application on Netlify to the Node Express server deployed and running on **Render** (or **Heroku**).

## 5.1 Committing and Pushing the Node Server Source to Github

First create a local Git repository in the Node Express project by typing **git init** at the command line at the root of the project. It's ok if the repository was already initialized.

```
$ git init
```

Configure the Git repository to disregard unnecessary files in the repository by listing them in **.gitignore**. Create a new file called **.gitignore** if it does not already exist. Note the leading period (.) in front of the file name. The file should contain at least **node\_modules**, but should also contain any IDE specific files or directories. If using IntelliJ, include the **.idea** folder as shown below. Environment files such as **.env** and **.env.local** should also be included in **.gitignore**.

```
.gitignore
node_modules
.env.local
```

Use **git add** to add all the source code into the repository and commit with a simple comment.

```
$ git add .
$ git commit -m "first commit"
```

Head over to **github.com** and create a repository named **kanbas-node-server-app**. Add this repository as the origin target using the git remote command. **NOTE:** make sure to use your github username instead of mine.

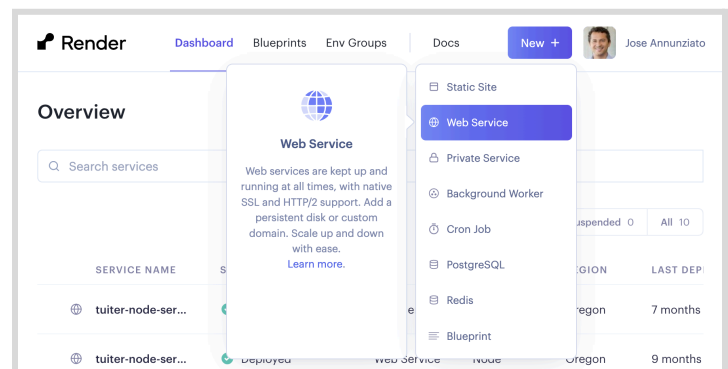
```
$ git remote add origin https://github.com/jannunzi/kanbas-node-server-app.git
```

Push the code in your local repository to the remote origin repository. Note that your branch might be called something else. Refresh the remote github repository and confirm the code is now available online.

```
$ git push -u origin main
```

## 5.2 Deploying a Node Server Application to Render.com from Github

If you don't already have an account at **render.com**, create a new account so we can deploy the Node server remotely. From the dashboard on the top right, select **New** and then **Web Service**. In the **Create a new Web Service** window select the option **Build and deploy from a Git repository** and click **Next**. In the **Create a new Web Service** screen, search for the GitHub repository you just created, e.g., **kanbas-node-server-app**. Click **Connect** on the correct repository from the list that appears. In the **You are deploying a web service** window, in the **Name** field, type the name of the application, e.g., use the same name as the github repository **kanbas-node-server-app**. In the **Build Command** field type **npm install**. In the **Start Command** field type **npm start**. Under the **Instance Type** select **Free**. Click **Create Web Service** to deploy the server. In the deploy screen take a look at the logs. You can click on **Maximize** to see the logs better. Look for a **Build successful** message. You can click on **Minimize** to minimize the logs. If the deployment fails, fix whatever the logs complaint about, commit and push your changes to GitHub, and try deploying again by selecting **Deploy last commit** from the **Manual Deploy** drop menu at the top right. If the deployment succeeds a URL appears at the top of the screen. Navigate to the URL <https://kanbas-node-server-app.onrender.com/api/courses> and confirm you get the same greeting you would get locally, e.g., **Welcome to Full Stack Development!** Also confirm you can get the array of courses from the remote API, e.g.,

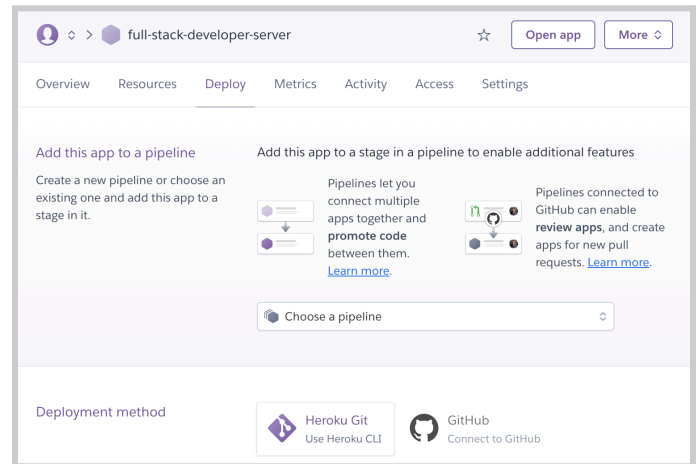




<https://kanbas-node-server-app.onrender.com/api/courses>. Finally, make sure you can get the list of modules for at least one of the courses, e.g., <https://kanbas-node-server-app.onrender.com/api/courses/RS101/modules>. **NOTE:** the actual URL might be different based on the actual name you chose for the application.

## 5.3 Deploying a Node Server Application to Heroku from Github

If you don't already have an account at [heroku.com](https://heroku.com), create a new account so we can deploy the Node server remotely. From the dashboard on the top right, select **New** and then **Create new app**. In the **app-name** field type the name of the application, e.g., use the same name as the github repository **kanbas-node-server-app**, and click **Create app**. If the name of the application is already taken, you'll need to try different variations. In the application dashboard select the **Deploy** tab and then the **GitHub Deployment method**. In the **Connect to GitHub** section, select the repository you wish to deploy from and then type the name of the repository in the **repo-name** field, e.g., **kanbas-node-server-app**. Click **Search** and then **Connect** on the correct repository from the list that appears. Under the **Automatic deploys** section, choose **master** or **main** under **Choose a branch to deploy** and then click **Enable Automatic Deploys** so that the server will automatically deploy whenever you push new code to the GitHub repository. You might not want to enable this feature and instead choose to deploy the server manually in the **Manual deploy** section, selecting the branch and then clicking **Deploy Branch**. If you click **Deploy Branch**, logs will scroll below displaying the process. If the deployment succeeds the message **Your app was successfully deployed** will appear. Click on **View** at the bottom of the page or on **Open App** at the top right and confirm the server responds with a welcoming message. Navigate to the URL listing the courses and confirm that they display correctly, e.g., <https://kanbas-node-server-app.herokuapp.com/api/courses>. **NOTE:** the actual URL might be different based on the actual name you chose for the application. In **TOC.tsx**, add a link to the new GitHub repository and a link to the root of the server running on **Render** or **Heroku**.



## 5.4 Configuring Remote Environments

The React Web application is configured to connect to the local Node Express server, but when the Web application is running on **Netlify** it needs to connect to the remote Node Express server running on **Render** or **Heroku**. Let's configure **Netlify** environment variables to define **REACT\_APP\_REMOTE\_SERVER** so that it references the remote server running on **Render** or **Heroku**.

To configure environment variables from the Netlify dashboard, navigate to **Site configuration**, then **Environment variables**, select **Add a variable**, and then select **Add a single variable**. In the **New environment variable** form here on the right, enter **REACT\_APP\_REMOTE\_SERVER** in the **Key** field, and then in the **Values** field, copy and paste the root URL of the application running on Render or Heroku, e.g., <https://kanbas-node-server-app.onrender.com>, and click **Create variable**. Redeploy the React application and confirm that the **Dashboard** renders the courses from the remote Node server and **Modules** still renders the modules for the selected course.

Also confirm all the labs still work when running on Netlify.

## 6 Conclusion

In this chapter we learned how to create HTTP servers using the Node.js JavaScript framework. We implemented RESTful services with the Express library and practiced sending, retrieving, modifying, and updating data using HTTP requests and responses. We then learned how to integrate React.js Web applications to HTTP servers implementing a client server architecture. In the next chapter we will add database support to the HTTP server so we can store data permanently.

## 7 Deliverables

As a deliverable, make sure you complete all the lab exercises, course, modules, and assignment routes on the server, as well as the client, and component refactoring on the React project. For both the React and Node repositories, all your work should be done in a branch called **a5**. When done, add, commit and push both branches to their respective GitHub repositories. Deploy the new branches to **Netlify** and **Render** (or **Heroku**) and confirm they integrate. All the lab exercises should work remotely just as well as locally. The Kanbas Dashboard should display the courses from the server as well as the modules, and assignments. In **TOC.tsx**, add a link to the new GitHub repository and a link to the root of the server running on **Render** or **Heroku**. As a deliverable in **Canvas**, submit the URL to the **a5** branch deployment of your React.js application running on Netlify.