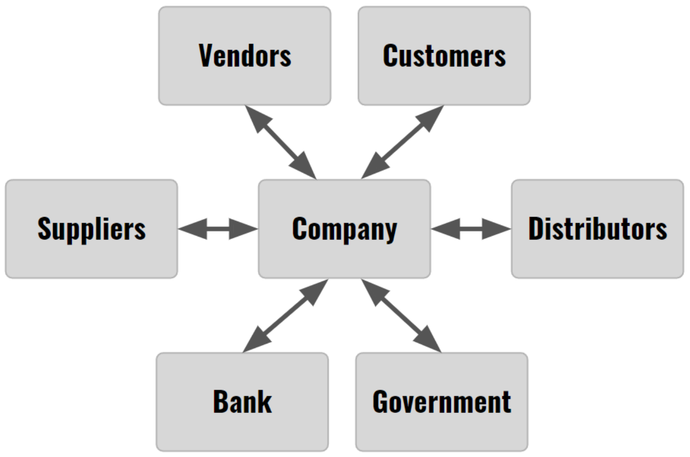
Implementing HTTP Servers and RESTful APIs with Node.js

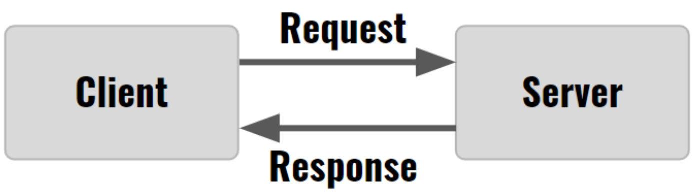
1 Introduction



During the 90s, the World Wide Web adoption grew exponentially. Various commercial ventures explored a plethora of use cases that revolutionized how companies interacted with their customers and with other companies. The figure on the right illustrates several integration points between businesses, sometimes referred as ***business to business*** or ***B2B***. Interactions between businesses and their customers is often referred to as ***business to consumer*** or ***B2C***. Companies have largely automated interactions with their customers by implementing online storefronts where customers can browse through products, order them, review them, and even return them. Interacting with users demands creating visually pleasing user interfaces that grab their attention, entice them to buy products with marketing ads, and establish a long term relation with their customers through incentives such as discounts and loyalty programs.

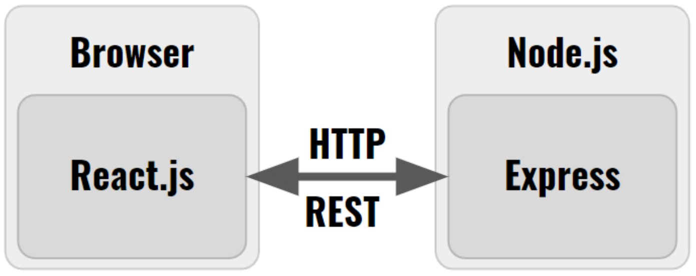
Thus far we've been focusing on implementing ***user interfaces*** which, as the name suggests, focus on the aspects of an application that interact with users through visually pleasing representations of some data set. The data rendered by these user interfaces has been, up to this point, hard coded JSON files, e.g., ***courses.json***, ***modules.json***. We learned how to create user interfaces that render and manipulate the data, and then update the screen to reflect the changes. Unfortunately the updates were not permanent, that is, if the browser window was refreshed, all the changes were lost and the state of the application was reset. In general JavaScript applications running on clients such as browsers, game consoles, or TV boxes, have limited options on how to retrieve and store data permanently. The next couple of chapter tackle 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 associated with as a programming language designed to execute in browsers, but Node.js has rescued it from its browser confines. Node.js is a JavaScript runtime that can interpret and execute applications written in JavaScript outside browsers, such as executing JavaScript from a desktop console or terminal. This is important because

JavaScript applications written for the desktop can compensate many of the limitations of JavaScript applications written for the browser. For instance JavaScript running on a browser doesn't have access to the filesystem, databases, and have restricted network access. On the other hand, JavaScript running on a desktop has unfettered access to the filesystem, databases, and full network access. Conversely desktop JavaScript applications don't generally have a user interface, and have limited user interaction whereas browser JavaScript applications can interact with users with rich sophisticated interfaces.

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 should already have been installed in your computer as you worked through the exercises in previous assignments, implementing the React.js Web application. Nevertheless it's useful to review and confirm you have a working Node.js installation. Using your computer terminal or console application, type the following to check the version of Node installed in your machine.

```
$ node  -v
v20.11.1
```

If there's a Node installation then its version will display on the console, otherwise there'll be an error message and you'll need to download and install Node.js from the URL below. As of this writing Node.js 20.11.1 was the latest version, but feel free to install whatever version is recommended on Node's Website.

```
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.

```
$ 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 -- not hugely important.

```
$ 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 the IDE of your choice, e.g, IntelliJ or Visual Studio Code, 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

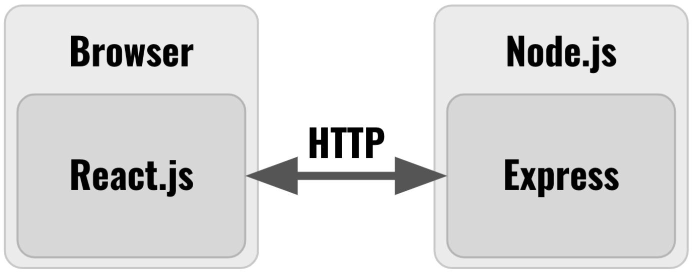
```
console.log("Hello World!");
```

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 will describe writing JavaScript applications that implement HTTP Web servers and RESTful APIs to integrate with React.js user interfaces. Upcoming chapters will describe writing JavaScript applications that store and retrieve data from NoSQL databases such as MongoDB.

2.5 Creating a Node.js HTTP Web server



Express is one of the most popular Node.js libraries simplifying creating HTTP servers. We'll use express to implement HTTP servers that can respond to HTTP requests from any HTTP client, but our React.js client in particular. From the root directory of the Node.js project, install the ***express*** library from the terminal as shown below.

```
$ npm  install  express
```

A new entry should appear 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. The following ***App.js*** implements an HTTP server that responds ***Hello World!*** when the server receives an HTTP request at the URL ***http://localhost:4000/hello***. You can 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***. We will use the ***app*** instance 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 handler*** by mapping the URL pattern ***'/hello'*** to a function that handles the HTTP request.

App.js

```
const express = require('express')                            // equivalent to import
const app = express()                                          // create new express instance
app.get('/hello', (req, res) => {res.send('Hello World!')})    // create a route that responds 'hello'
app.listen(4000)                                               // 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  App.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***. From the point of view of browsers, ***http://localhost:4000/hello*** is referred to as a ***URL*** (***Uniform Resource Locator***). From the point of view of the server we often use the term ***HTTP endpoint*** or just ***endpoint***.

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  App.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.

App.js

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

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 ***App.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 App.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 ***App.js*** refactored to use ***import*** instead of ***require***. Refresh the browser and confirm that the server responds as expected.

App.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);
```

```
// now we can use import syntax instead of require
```

2.8 Creating HTTP Routes

The ***App.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 chapters we're going to create quite a few other HTTP routes, too many to define them all in ***App.js***. Let's move both HTTP routes to the ***Hello.js*** file created earlier as shown bellow.

App.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 App.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 ***App.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.

App.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. Create Import file ***Lab5.js*** where we will be implementing several server side exercises. Create a route that welcomes users to Assignment 5.

Lab5.js

```
const Lab5 = (app) => {
  app.get("/a5/welcome", (req, res) => {
    res.send("Welcome to Assignment 5");
  });
};
export default Lab5;
```

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

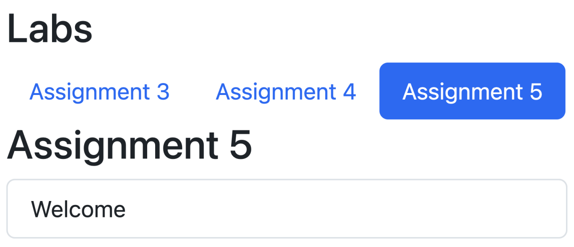
Import ***Lab5*** into the server ***App.js*** and pass it a reference to ***express*** as shown below. Restart the server and confirm that http://localhost:4000/a5/welcome responds with the expected response.

App.js

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

```

// import Lab5


// pass reference to express module
```

Now let's create an ***Assignment5*** React.js component to test the Node HTTP server. Import the new component into your existing set of labs so that you can navigate to it by selecting the corresponding tab or link as shown below. The example below creates a hyperlink that navigates to the http://localhost:4000/a5/welcome URL. Confirm that link navigates to the expected response.



src/Labs/a5/index.ts

```
function Assignment5() {
  return (
    <div>
      <h1>Assignment 5</h1>
      <a href="http://localhost:4000/a5/welcome">
        Welcome
      </a>
    </div>
  );
}
export default Assignment5;
```

```
// hyperlink navigates to
// http://localhost:4000/a5/welcome
```

3.1 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/a5/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/a5/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.

3.1.1 Path Parameters

React.js applications can pass data to servers by embedding it in a URL path as ***path parameters*** part of the 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/a5/add/2/4

The following route declarations can parse path parameters a and b encoded in paths ***/a5/add/:a/:b*** and ***/a5/subtract/:a/:b***. Implement the routes in ***Lab5.js*** in your Node application and confirm that http://localhost:4000/a5/add/6/4 responds with 10 and http://localhost:4000/a5/subtract/6/4 responds with 2.

Lab5.js

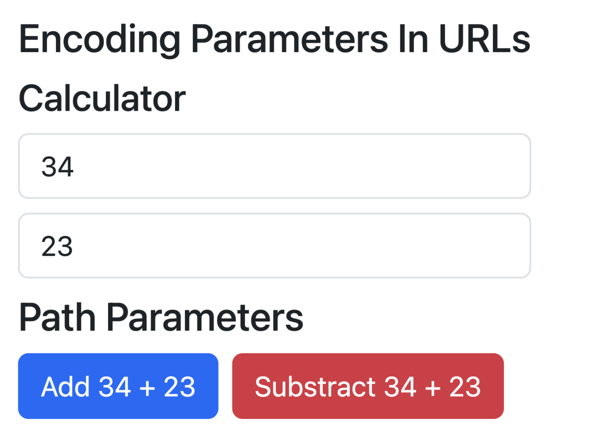
```
const Lab5 = (app) => {
  app.get("/a5/welcome", (req, res) => {
    res.send("Welcome to Assignment 5");
  });
  app.get("/a5/add/:a/:b", (req, res) => {
    const { a, b } = req.params;
    const sum = parseInt(a) + parseInt(b);
    res.send(sum.toString());
  });
  app.get("/a5/subtract/:a/:b", (req, res) => {
    const { a, b } = req.params;
    const sum = parseInt(a) - parseInt(b);
    res.send(sum.toString());
  });
};
export default Lab5;
```

```
// route expects 2 path parameters after /a5/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 /a5/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
```

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.

src/Labs/a5/EncodingParametersInURLs.ts

```
import React, { useState } from "react";
function EncodingParametersInURLs() {
  const [a, setA] = useState(34);
  const [b, setB] = useState(23);
  return (
    <div>
      <h3>Encoding Parameters In URLs</h3>
      <h4>Calculator</h4>
      <input type="number" value={a}
        onChange={(e) => setA(e.target.value)}/>
      <input type="number"
        onChange={(e) => setB(e.target.value)} value={b}/>
      <h3>Path Parameters</h3>
      <a href={`http://localhost:4000/a5/add/${a}/${b}`}>
        Add {a} + {b}
      </a>
      <a href={`http://localhost:4000/a5/subtract/${a}/${b}`}>
        Substract {a} - {b}
      </a>
    </div>
  );
}
export default EncodingParametersInURLs;
```



Include the new component in your ***Assignment5*** component and confirm that clicking the links generates the expected response.

3.1.2 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/a5/add?a=2&b=4

In ***Lab5.js***, 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.

Lab5.js

```
  ...
  app.get("/a5/calculator", (req, res) => {          // e.g., a5/calculator?a=5&b=2&operation=add
    const { a, b, operation } = req.query;           // retrieve a, b, and operation parameters in query
    let result = 0;
    switch (operation) {
      case "add":
        result = parseInt(a) + parseInt(b);          // parse as integers since parameters are strings
        break;
      case "subtract":
        result = parseInt(a) - parseInt(b);
        break;
      default:
        result = "Invalid operation";
    }
    res.send(result.toString());                     // convert to string otherwise browser interprets
  });                                                // as a status code
```

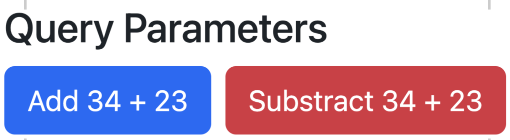
In the ***EncodingParametersInURLs*** component, create two additional hyperlinks to test the new route as shown below. Confirm the following hyperlinks work as expected.

Test Hyperlinks	***Confirm Response***
http://localhost:4000/a5/calculator?operation=add&a=34&b=23	57
http://localhost:4000/a5/calculator?operation=subtract&a=34&b=23	11

src/Labs/a5/EncodingParametersInURLs.ts

```
...
<h3>Query Parameters</h3>
<a className="btn btn-primary"
  href={`http://localhost:4000/a5/calculator?operation=add&a=${a}&b=${b}`}>
  Add {a} + {b}
</a>
<a className="btn btn-danger"
  href={`http://localhost:4000/a5/calculator?operation=subtract&a=${a}&b=${b}`}>
  Substract {a} - {b}
</a>
...
```

```
//
```



3.1.3 On Your Own

Now, on your own, 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***.

3.2 Working with Objects

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 ***/a5/assignment***.

Lab5.js

```
const assignment = {
  id: 1, title: "NodeJS Assignment",
  description: "Create a NodeJS server with ExpressJS",
  due: "2021-10-10", completed: false, score: 0,
};
const Lab5 = (app) => {
  app.get("/a5/assignment", (req, res) => {
    res.json(assignment);
  });
  ...
};
export default Lab5;
```

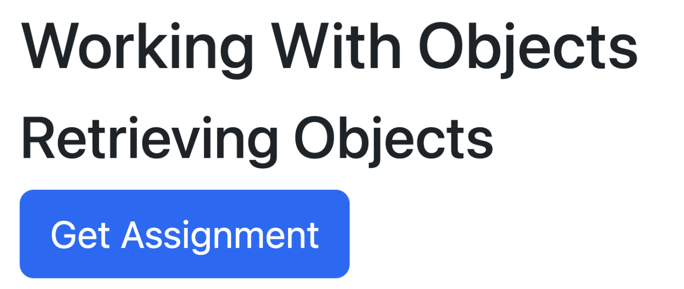
```
// object state persists as long
// as server is running
// changes to the object persist
// rebooting server
// resets the object


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

3.2.1 Retrieving Objects from a Server

Create ***WorkingWithObjects*** component to test the new route as shown below. Include in ***Assignment5*** and confirm that http://localhost:4000/a5/assignment responds with ***assignment*** object.

src/Labs/a5/WorkingWithObjects.ts

```
import React, { useState } from "react";
function WorkingWithObjects() {
  return (
    <div>
      <h3>Working With Objects</h3>
      <h4>Retrieving Objects</h4>
      <a href="http://localhost:4000/a5/assignment">
        Get Assignment
      </a>
    </div>
  );
}
export default WorkingWithObjects;
```


Working With Objects

Retrieving Objects

3.2.2 Retrieving Object Properties from a Server

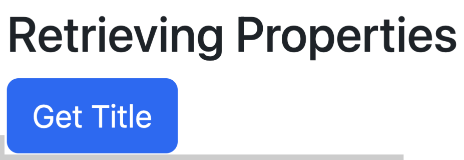
We can retrieve individual properties in an object such as the ***title*** shown below.

Lab5.js

```
...
const Lab5 = (app) => {
  app.get("/a5/assignment", (req, res) => {
    res.json(assignment);
  });
  app.get("/a5/assignment/title", (req, res) => {
    res.json(assignment.title);
  });
  ...
};
export default Lab5;
```

```
// respond with string property
// can do the same with other properties
```

Confirm that the http://localhost:4000/a5/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.


Retrieving Properties

src/Labs/a5/WorkingWithObjects.ts

```
      ...
      <h4>Retrieving Objects</h4>
      <a href="http://localhost:4000/a5/assignment">
        Get Assignment
      </a>
      <h4>Retrieving Properties</h4>
      <a href="http://localhost:4000/a5/assignment/title">
        Get Title
      </a>
      ...
```

3.2.3 Modifying Objects in a Server

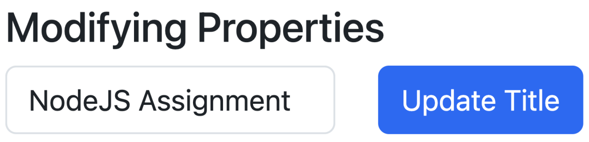
We can also use routes to modify objects or individual properties as shown below.

Lab5.js

```
  ...
  app.get("/a5/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
```

Create another route mapped to ***/a5/module/name/:newName*** that can change the ***name*** of the ***module***. In ***WorkingWithObjects***, 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

NodeJS Assignment

src/Labs/a5/WorkingWithObjects.ts

```
import React, { useState } from "react";
function WorkingWithObjects() {
  const [assignment, setAssignment] = useState({      // create a state variable that holds
    id: 1, title: "NodeJS Assignment",                // default values for the form below.
    description: "Create a NodeJS server with ExpressJS",  // eventually we'll fetch this initial
    due: "2021-10-10", completed: false, score: 0,    // data from the server and populate
  });                                                  // the form with the remote data so
  const ASSIGNMENT_URL = "http://localhost:4000/a5/assignment"  // we can modify it here in the UI
  return (
    <div>
      <h3>Working With Objects</h3>
      <h4>Modifying Properties</h4>
      <a href={`${ASSIGNMENT_URL}/title/${assignment.title}`}>   // encode the title in the URL that
        Update Title                                              // updates the title
      </a>
      <input type="text"
        onChange={(e) => setAssignment({ ...assignment,          // form element to edit local state variable
            title: e.target.value })}                             // used to encode in URL that updates property
        value={assignment.title}/>                                // in remote object
      ...
    </div>
  );
}
export default WorkingWithObjects;
```

3.2.4 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 ***/a5/module***
- In the UI, create a link labeled ***Get Module*** that retrieves the ***module*** object from the server mapped at ***/a5/module***
- Confirm that clicking the link retrieves the module.
- Create another route mapped to ***/a5/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.ts***, 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.

3.3 Working with Arrays

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.

3.3.1 Retrieving Arrays

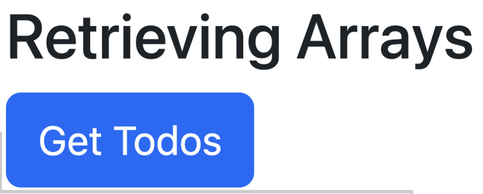
Let's First create the array and create the route below to retrieve the array.

Lab5.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 },
];
const Lab5 = (app) => {
  app.get("/a5/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 ***Assignment5*** complement and confirm clicking the link retrieves the array.



src/Labs/a5/WorkingWithArrays.ts

```
function WorkingWithArrays() {
  const API = "http://localhost:4000/a5/todos";
  return (
    <div>
      <h3>Working with Arrays</h3>
      <h4>Retrieving Arrays</h4>
      <a href={API}>
        Get Todos
      </a>
    </div>
  );
}
export default WorkingWithArrays;
```

3.3.2 Retrieving an Item from an Array by ID

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.js

```
  ...
  app.get("/a5/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/a5/WorkingWithArrays.ts

```
import React, { useState } from "react";
function WorkingWithArrays() {
      ...
      const [todo, setTodo] = useState({id: 1});
      ...
      <h4>Retrieving Arrays</h4>
      <a href={API}>
        Get Todos
      </a>
      <h4>Retrieving an Item from an Array by ID</h4>
      <input value={todo.id}
        onChange={(e) => setTodo({ ...todo,
          id: e.target.value })}/>
      <a href={`${API}/${todo.id}`}>
        Get Todo by ID
      </a>
      ...
```

```
//
```

3.3.3 Filtering array items using 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 ***/a5/todos*** to handle the case when we want to filter the array by the ***completed*** query parameter.

Lab5.js

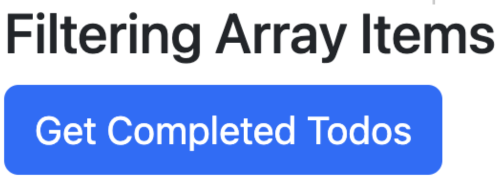
```
const todos = [
  { id: 1, title: "Task 1", completed: false },
  { id: 2, title: "Task 2", completed: true },
  { id: 3, title: "Task 3", completed: false },
  { id: 4, title: "Task 4", completed: true },
];
const Lab5 = (app) => {
  app.get("/a5/todos", (req, res) => {
    const { completed } = req.query;
    if (completed !== undefined) {
      const completedBool = completed === "true";
      const completedTodos = todos.filter(
        (t) => t.completed === completedBool);
      res.json(completedTodos);
      return;
    }
    res.json(todos);
  });
  ...
};
export default Lab5;
```

```
//
```

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

src/Labs/a5/WorkingWithArrays.ts

```
...
<h3>Retrieving Arrays</h3>
...
<h3>Retrieving an Item from an Array by ID</h3>
<a href={`${API}/${todo.id}`}>
  Get Todo by ID
</a>
<h3>Filtering Array Items</h3>
<a href={`${API}?completed=true`}>
  Get Completed Todos
</a>
...
```



3.3.4 Creating new Items in an Array

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 ***/a5/todos/:id*** route, otherwise the ***:id*** path parameter would interpret the "***create***" in ***/a5/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.js

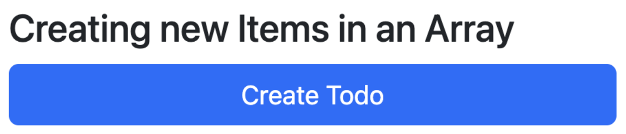
```
...
app.get("/a5/todos/create", (req, res) => {
  const newTodo = {
    id: new Date().getTime(),
    title: "New Task",
    completed: false,
  };
  todos.push(newTodo);
  res.json(todos);
});
app.get("/a5/todos/:id", (req, res) => {
  const { id } = req.params;
  const todo = todos.find((t) => t.id === parseInt(id));
  res.json(todo);
});
...
```

```
// make sure to implement this BEFORE the /a5/todos/:id
// route implemented below

// make sure to implement this AFTER the
// /a5/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/a5/WorkingWithArrays.ts

```
...
<h3>Creating new Items in an Array</h3>
<a href={`${API}/create`}>
  Create Todo
</a>
...
```



3.3.5 Deleting an Item from an Array

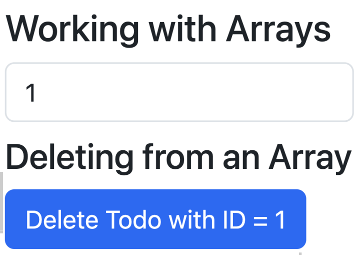
Next let's consider the ***delete*** operation in the ***CRUD*** family of operations. The convention is to encode the ID of the item to delete as a path parameter as shown below. We search for the item in the set of items and remove it. Typically we would respond with a status of success or failure,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

Lab5.js

```
  ...
  app.get("/a5/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);
  });
  ...
```

```
//
```

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.


Working with Arrays

1

Deleting from an Array

Delete Todo with ID = 1

src/Labs/a5/WorkingWithArrays.ts

```
      ...
      <h3>Deleting from an Array</h3>
      <a href={`${API}/${todo.id}/delete`}>
        Delete Todo with ID = {todo.id}
      </a>
      ...
```

3.3.6 Updating an Item in an Array

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.js

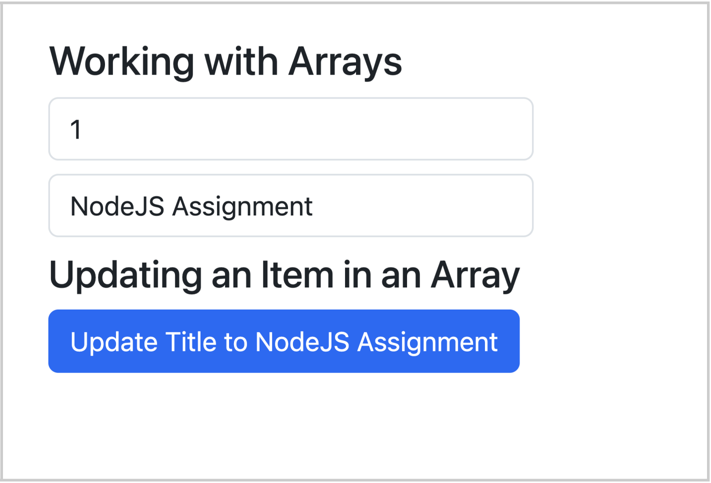
```
...
  app.get("/a5/todos/:id/title/:title", (req, res) => {
    const { id, title } = req.params;
    const todo = todos.find((t) => t.id === parseInt(id));
    todo.title = title;
    res.json(todos);
  });
...
```

```
//
```

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/a5/WorkingWithArrays.ts

```
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>
      <input type="number" value={todo.id}
        onChange={(e) => setTodo({
          ...todo, id: e.target.value })}/>
      <input type="text" value={todo.title}
        onChange={(e) => setTodo({
          ...todo, title: e.target.value })}/>
      <h3>Updating an Item in an Array</h3>
      <a href={`${API}/${todo.id}/title/${todo.title}`} >
        Update Title to {todo.title}
      </a>
...
```


Working with Arrays

1

NodeJS Assignment

Updating an Item in an Array

3.3.7 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.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	***/a5/todos/:id/completed/:completed***	todos	Complete Todo ID = 1
description	***/a5/todos/:id/description/:description***	todos	Describe Todo ID = 1

3.4 Receiving Data from Servers as HTTP Responses

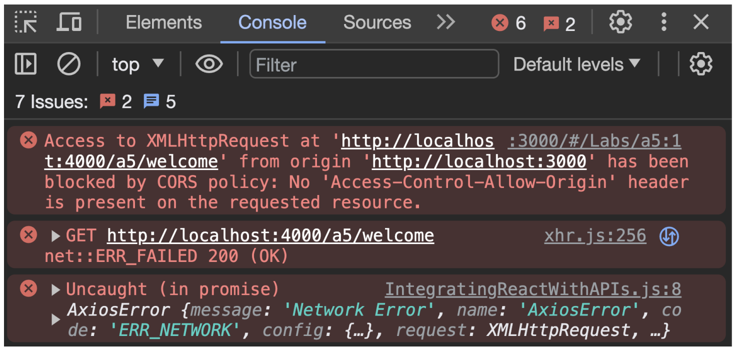
The exercises explored so far sent data encoded in the URL of hyperlinks. The links navigated to a separate browser window that displayed the server response. Even though we were able to send data to the server and affect changes to the server data, we have not considered how those changes can affect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

3.4.1 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 application 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 routes implemented in earlier exercises, but instead of clicking on hyperlinks, ***axios*** will programmatically invoke the URLs giving us a chance to capture and handle the responses from the server. The ***useEffect*** hook configures a function to call when the component first loads. In the example below, ***useEffect*** invokes the ***fetchWelcome*** when the component loads. The ***fetchWelcome*** function asynchronous uses ***axios*** to send a ***GET*** request to ***/a5/welcome*** and returns the ***response*** from the server. The data from the server is available in ***response.data*** which we store in state variable ***welcome***. Copy the code below into ***EncodingParametersInURLs.ts*** and refresh the browser. Open the console and confirm you get a ***CORS*** error similar to the one here on the right. We'll fix this error in the next section.

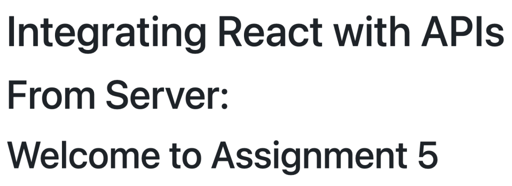


```
Access to XMLHttpRequest at 'http://localhos :3000/#/Labs/a5:1
t:4000/a5/welcome' from origin 'http://localhost:3000' has been
blocked by CORS policy: No 'Access-Control-Allow-Origin' header
is present on the requested resource.
▶ GET http://localhost:4000/a5/welcome                xhr.js:256
net::ERR_FAILED 200 (OK)
▶ Uncaught (in promise)              IntegratingReactWithAPIs.js:8
 ▶ AxiosError {message: 'Network Error', name: 'AxiosError', co
  de: 'ERR_NETWORK', config: {…}, request: XMLHttpRequest, …}
```

src/Labs/a5/EncodingParametersInURLs.ts

```
import React, { useEffect, useState } from "react";
import axios from "axios";
function EncodingParametersInURLs() {
  const [a, setA] = useState(34);
  const [b, setB] = useState(23);
  const [welcome, setWelcome] = useState("");
  const fetchWelcome = async () => {
    const response = await axios.get("http://localhost:4000/a5/welcome");
    setWelcome(response.data);
  };
  useEffect(() => {
    fetchWelcome();
  }, []);
  return (
    <div>
      <h3>Encoding Parameters In URLs</h3>
      <h4>Integrating React with APIs</h4>
      <h5>Fetching Welcome</h5>
      <h6>{welcome}</h6>
      ...
    </div>
  );
}
export default EncodingParametersInURLs;
```

3.4.2 Configuring Cross Origin Request Sharing (CORS)



Integrating React with APIs

From Server:

Welcome to Assignment 5

In the early history of the Web, JavaScript got a bad rap and was considered a security risk. To address these concerns, servers and browsers agreed to limit JavaScript programs to only be able to communicate with the servers from where they had been downloaded from. Since our Reac.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 and 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 ***App.js***, import the ***cors*** library and configure it as shown below. Restart the server and refresh the React.js application. Confirm that the user interface is able to retrieve the ***Welcome to Assignment 5*** message from the server without errors.

App.js

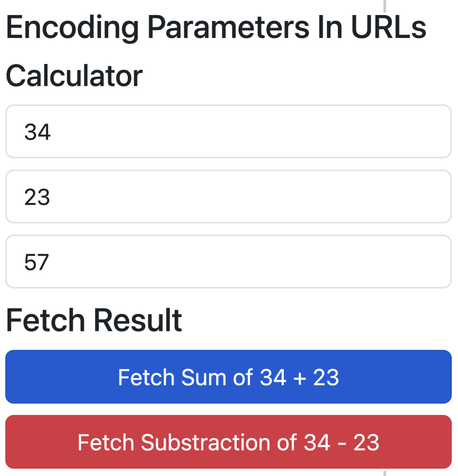
```
import express from "express";
import Lab5 from "./Lab5.js";
import cors from "cors";
const app = express();
app.use(cors());                 // make sure cors is used right after creating the app
Lab5(app);                       // express instance
app.listen(4000);
```

3.4.3 Sending and fetching data to and from an HTTP server

Let's revisit some of the same exercises we worked on earlier, but this time using ***axios.get*** to send the same requests and capturing the responses so we can update the user interface. Add a ***result*** state variable where we can grab the responses to the ***add*** and ***subtract*** requests as shown below. Add buttons to invoke the new fetch functions that calculate the addition and subtraction of the input fields for ***a*** and ***b***. Confirm that clicking the two new buttons calculates the result correctly.

src/Labs/a5/EncodingParametersInURLs.ts

```
import axios from "axios";
function EncodingParametersInURLs() {
  const [a, setA] = useState(34);
  const [b, setB] = useState(23);
  const [result, setResult] = useState(0);
  const fetchSum = async (a, b) => {
    const response = await
      axios.get(`http://localhost:4000/a5/add/${a}/${b}`);
    setResult(response.data);
  };
  const fetchSubtraction = async (a, b) => {
    const response = await axios.get(
      `http://localhost:4000/a5/subtract/${a}/${b}`);
    setResult(response.data);
  };
  useEffect(() => { ... }, []);
  return (
    <div>
      ...
      <h4>Calculator</h4>
      <input ... value={a} ... />
      <input ... value={b} ... />
      <input value={result} type="number" readOnly />
      <h3>Fetch Result</h3>
      <button onClick={() => fetchSum(a, b)} >
        Fetch Sum of {a} + {b}
      </button>
      <button onClick={() => fetchSubtraction(a, b)} >
        Fetch Substraction of {a} - {b}
      </button>
      ...
    </div>
  );
}
```


Encoding Parameters In URLs

Calculator

34

23

57

Fetch Result

3.4.4 Fetching and updating objects

Let's now revisit the APIs that worked with the ***assignment*** object in ***WorkingWithObjects***. Add functions that fetch the assignment object and update its title, and update the state variable with the response as shown below. Confirm that you can update the assignment's title as well as fetch the assignment. Note how this new implementation fetches and modifies the remote object without navigating away from the user interface. Communication with the server happens befind the scenes and the user interface can update dynamically.

src/Labs/a5/WorkingWithObjects.ts

```
import React, { useEffect, useState } from "react";
import axios from "axios";
function WorkingWithObjects() {
  const ASSIGNMENT_URL = "http://localhost:4000/a5/assignment";
  const fetchAssignment = async () => {
    const response = await axios.get(`${ASSIGNMENT_URL}`);
    setAssignment(response.data);
  };
  const updateTitle = async () => {
    const response = await axios
      .get(`${ASSIGNMENT_URL}/title/${assignment.title}`);
    setAssignment(response.data);
  };
  useEffect(() => {
    fetchAssignment();
  }, []);
  return (
    <div>
      <h2>Working With Objects</h2>
      <h3>Modifying Properties</h3>
      <input onChange={(e) => setAssignment({
            ...assignment, title: e.target.value })}
        value={assignment.title} type="text" />
      <button onClick={updateTitle} >
        Update Title to: {assignment.title}
      </button>
      <button onClick={fetchAssignment} >
        Fetch Assignment
      </button>
    </div>
  );
}
```

```
//
```

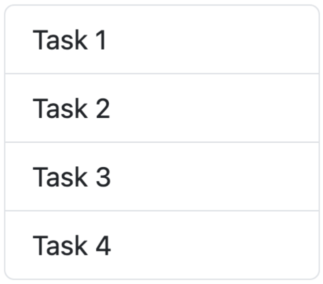
3.4.5 Fetching Arrays

The examples that worked with arrays can also be enhanced with ***axios***. 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/a5/WorkingWithArrays.ts

```
import React, { useState, useEffect } from "react";
import axios from "axios";
function WorkingWithArrays() {
  const API = "http://localhost:4000/a5/todos";
  const [todo, setTodo] = useState({ ... });
  const [todos, setTodos] = useState<any[]>([]);
  const fetchTodos = async () => {
    const response = await axios.get(API);
    setTodos(response.data);
  };
  useEffect(() => {
    fetchTodos();
  }, []);
  return (
    <div>
      <h2>Working with Arrays</h2>
      ...
      <ul>
        {todos.map((todo) => (
          <li key={todo.id}>
            {todo.title}
          </li>
        ))}
      </ul>
    </div>
  );}
```

```
//
```

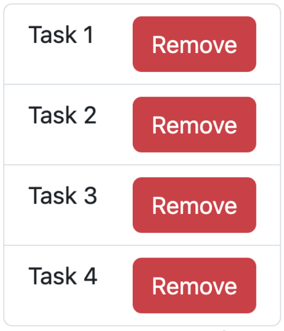

Task 1
Task 2
Task 3
Task 4

3.4.6 Deleting items from an array

Add ***Remove*** buttons to the todos that invoke a new ***removeTodo*** function that sends an asynchronous delete request to the server and updates the ***todos*** state variable with the surviving todos. Confirm that clicking on the new ***Remove*** buttons actually removes the corresponding todo.

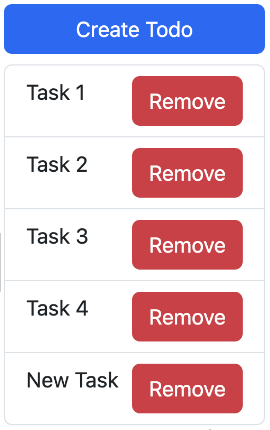
src/Labs/a5/WorkingWithArrays.ts

```
  ...
  const fetchTodos = async () => {
    const response = await axios.get(API);
    setTodos(response.data);
  };
  const removeTodo = async (todo) => {
    const response = await axios
      .get(`${API}/${todo.id}/delete`);
    setTodos(response.data);
  };
  ...
  return (
  ...
  <ul>
    {todos.map((todo) => (
      <li key={todo.id}>
        <button onClick={() => removeTodo(todo)} >
          Remove
        </button>
        {todo.title}
      </li>
    ))}
  </ul>
  ...
```



3.4.7 Create new array elements

Add a new ***Create Todo*** button that invokes a new ***createTodo*** function that sends an asynchronous create request to the server and updates the ***todos*** state variable with the new todo. Confirm that clicking the new ***Create Todo*** buttons actually adds a new todo item.



src/Labs/a5/WorkingWithArrays.ts

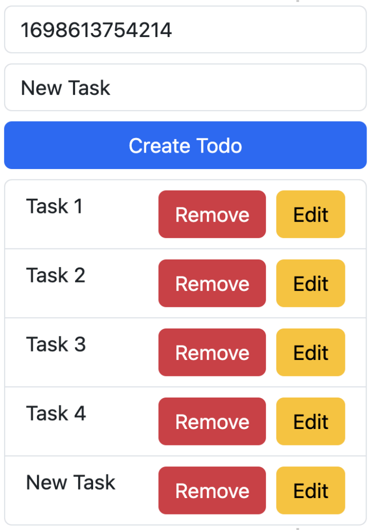
```
  ...
  const createTodo = async () => {
    const response = await axios.get(`${API}/create`);
    setTodos(response.data);
  };
  ...
  return (
    <div>
      <h2>Working with Arrays</h2>
      ...
      <button onClick={createTodo} >
        Create Todo
      </button>
      ...
    </div>
  );}
```

3.4.8 Fetching an item by its primary key ID

Add ***Edit*** buttons to the todos that invoke a new ***fetchTodoById*** function that sends an asynchronous request to the server for the todo item with the matching ID and updates the ***todo*** state variable. Confirm that clicking on the new ***Edit*** buttons actually fetches the corresponding todo and displays it in the form elements above.

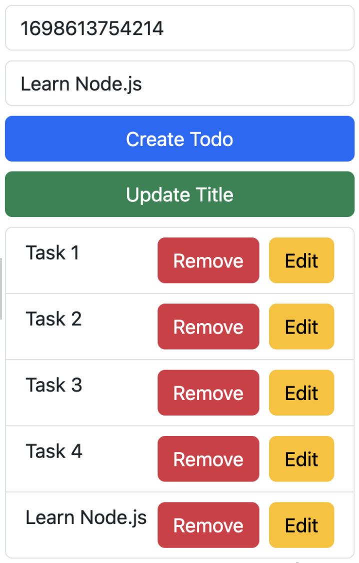
src/Labs/a5/WorkingWithArrays.ts

```
...
const [todo, setTodo] = useState({ ... });
...
const removeTodo = async (todo) => { ... }
const fetchTodoById = async (id) => {
  const response = await axios.get(`${API}/${id}`);
  setTodo(response.data);
};
...
<input value={todo.id}    onChange={(e) => setTodo({ ... })} />
<input value={todo.title} onChange={(e) => setTodo({ ... })} />
<ul className="list-group">
  {todos.map((todo) => (
    <li key={todo.id} className="list-group-item">
      <button onClick={() => fetchTodoById(todo.id)} >
        Edit
      </button>
      <button ... >
        Remove
      </button>
      {todo.title}
    </li>
  ))}
</ul>
...
```



3.4.9 Update array elements

Add a new ***Update Title*** button that invokes a new ***updateTitle*** function that sends an asynchronous request that updates the title of the currently selected todo in the UI. Then the todos are updated in the ***todos*** state variable and they rerender on the screen. Confirm that clicking the new ***Update Title*** buttons actually updates the title of the currently selected todo.



src/Labs/a5/WorkingWithArrays.ts

```
...
const updateTitle = async () => {
  const response = await axios.get(`${API}/${todo.id}/title/${todo.title}`);
  setTodos(response.data);
};
...
return (
  <div>
    <h2>Working with Arrays</h2>
    <button onClick={createTodo}> Create Todo </button>
    <input  value={todo.id}    onChange={ ... } />
    <input  value={todo.title} onChange={ ... } />
    <button onClick={updateTitle} >
      Update Title
    </button>
    ...
  </div>
);
}
```

3.5 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 personal data. A better approach is to encode the data as JSON in the HTTP's body which allows for arbitrarily large amounts of data as well as secure data encryption. Configure the Node.js server to parse JSON from the body of incoming HTTP requests as shown below.

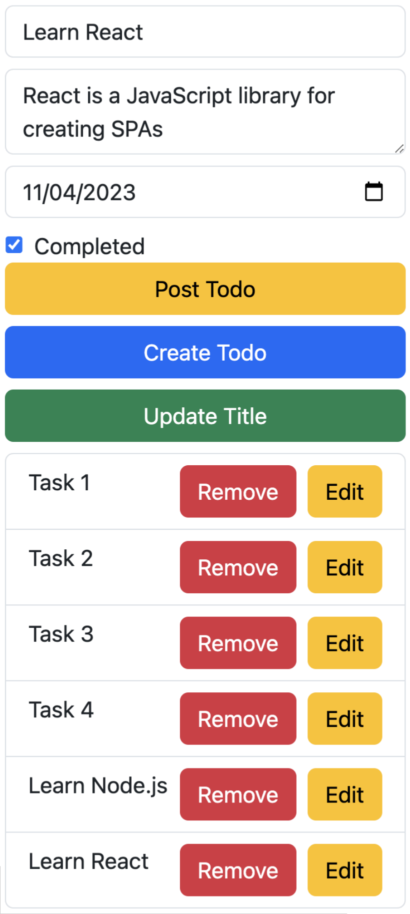
App.js	

```
import Lab5 from "./Lab5.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***. 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 personal data. A better approach is to encode the data as JSON in the HTTP's body which allows for arbitrarily large amounts of data as well as secure data encryption. Configure the Node.js server to parse JSON from the body of incoming HTTP requests as shown below. 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.


Learn React

React is a JavaScript library for creating SPAs

11/04/2023

Completed

3.5.1 Posting data in an HTTP Body

To illustrate using the HTTP POST method, let's re-implement the route that creates new todos as shown below. The convention is that URL paths should only include nouns and no verbs. The HTTP method POST 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 red 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 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.js	

```
const todos = [ ... ];
const Lab5 = (app) => {
  app.post("/a5/todos", (req, res) => {
    const newTodo = {
      ...req.body,
      id: new Date().getTime(),
    };
    todos.push(newTodo);
```
	`//`

```
    res.json(newTodo);
  });
  app.get("/a5/todos/create", (req, res) => {
    const newTodo = {
      id: new Date().getTime(),
      title: "New Task", completed: false,
    };
    todos.push(newTodo);
    res.json(todos);
  });
  app.get("/a5/todos", (req, res) => {
    res.json(todos);
  });
  ...
};
```

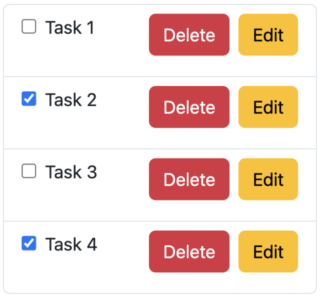
Back in the user interface, let's add a new ***postTodo*** function that posts new objects to the server as shown below. Note the second argument in the ***axios.post()*** method containing the new ***todo*** object instance sent to the server. The response this time contains the todo instance added to the array instead of all the todos on the server. Instead we reuse the todos already in the ***todos*** state variable to append the new ***todo*** from the server response at the end of the ***todos*** state variable. Confirm that you can add new items to the array when you click on the new ***Post Todo*** button.

src/Labs/a5/WorkingWithArrays.ts

```
function WorkingWithArrays() {
  const API = "http://localhost:4000/a5/todos";
  const [todo, setTodo] = useState({ ... });
  const [todos, setTodos] = useState([]);
  const postTodo = async () => {
    const response = await axios.post(API, todo);
    setTodos([...todos, response.data]);
  };
  useEffect(() => { ... }, []);
  return (
    <div>
      <h2>Working with Arrays</h2>
      <input value={todo.id} readOnly />
      <input onChange={(e) => setTodo({ ... })}
        value={todo.title} type="text" />
      <textarea value={todo.description} type="text"
        onChange={(e) => setTodo({ ...todo,
          description: e.target.value })} />
      <input value={todo.due} type="date"
        onChange={(e) => setTodo({
          ...todo, due: e.target.value })} />
      <label>
        <input value={todo.completed} type="checkbox"
          onChange={(e) => setTodo({
            ...todo, completed: e.target.checked })} />
        Completed
      </label>
      <button onClick={postTodo}> Post Todo </button>
      <ul className="list-group">
        {todos.map((todo) => (
          <li key={todo.id} className="list-group-item">
            <input checked={todo.completed}
              type="checkbox" readOnly />
            {todo.title}
            <p>{todo.description}</p>
            <p>{todo.due}</p>
            <button onClick={() => fetchTodoById(todo.id)} >
              Edit
            </button>
            <button onClick={() => removeTodo(todo)} >
              Remove </button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

```
//
```

3.5.2 Deleting Data



Following the convention that URL paths should include only nouns and no verbs, let's reimplement the route that deletes items from the array as shown below. Instead of having the ***delete*** verb in the previous route, the new implementation uses the HTTP delete verb to handle removing the item by its ID. 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 filter the item from the state variable.

Lab5.js

```
...
app.delete("/a5/todos/:id", (req, res) => {
  const { id } = req.params;
  const todo = todos.find((t) => t.id === parseInt(id));
  todos.splice(todos.indexOf(todo), 1);
  res.sendStatus(200);
});
app.get("/a5/todos/:id/delete", (req, res) => {
  const { id } = req.params;
  const todo = todos.find((t) => t.id === parseInt(id));
  todos.splice(todos.indexOf(todo), 1);
  res.json(todos);
});
...
```

```
//
```

Back in the user interface, create a new ***deleteTodo*** asynchronous function that invokes ***axios.delete*** passing the ID of the item to be removed from the server array. Then it filters out the item from the local ***todos*** state variable. Add a new ***Delete*** button to each of the todo items that invokes the new ***deleteTodo*** function passing the ***todo*** instance to be removed. Confirm that clicking this new ***Delete*** button actually removes the item from the server array.

src/Labs/a5/WorkingWithArrays.ts

```
...
const deleteTodo = async (todo) => {
  const response = await axios.delete(`${API}/${todo.id}`);
  setTodos(todos.filter((t) => t.id !== todo.id));
};
...
<button onClick={() => deleteTodo(todo)}
  className="btn btn-danger float-end ms-2">
  Delete
</button>
...
```

```
//
```

3.5.3 Updating Todo

Following the convention that URL paths should include only nouns and no verbs, let's reimplement the route that updates an item in an array as shown below. Instead of having the specific property we are modifying such as ***title***, ***completed***, or ***due***, like in previous routes, the new implementation uses a single ***HTTP put*** verb to implement updating the item by its ID. The updates are embedded in the HTTP body from which we copy the new values onto the old todo. 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.js

```
  app.put("/a5/todos/:id", (req, res) => {
    const { id } = req.params;
    const todo = todos.find((t) => t.id === parseInt(id));
    todo.title = req.body.title;
    todo.description = req.body.description;
    todo.due = req.body.due;
    todo.completed = req.body.completed;
    res.sendStatus(200);
  });
```

//

Back in the user interface, let's 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 this time contains a status instead of all the todos on the server. Instead we reuse the todos already in the ***todos*** state variable to replace the corresponding todo with the new ***todo***. Confirm that you can update items to the array when you click on the new ***Update Todo*** button.

src/Labs/a5/WorkingWithArrays.ts

```
import axios from "axios";
function WorkingWithArrays() {
  const API = "http://localhost:4000/a5/todos";
  ...
  const updateTodo = async () => {
    const response = await axios.put(`${API}/${todo.id}`, todo);
    setTodos(todos.map((t) => (t.id === todo.id ? todo : t)));
  };
  ...
  return (
    <div>
      ...
      <button onClick={postTodo}>
        Post Todo
      </button>
      <button onClick={updateTodo}>
        Update Todo
      </button>
      ...
    </div>
  );
}
export default WorkingWithArrays;
```

//

3.5.4 Handling Errors (graduates only)

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.js

```
  app.delete("/a5/todos/:id", (req, res) => {
    const { id } = req.params;
    const todo = todos.find((t) => t.id === parseInt(id));
    if (!todo) {
      res.status(404)
        .json({ message: `Unable to delete Todo with ID ${id}` });
      return;
    }
    todos.splice(todos.indexOf(todo), 1);
    res.sendStatus(200);
```

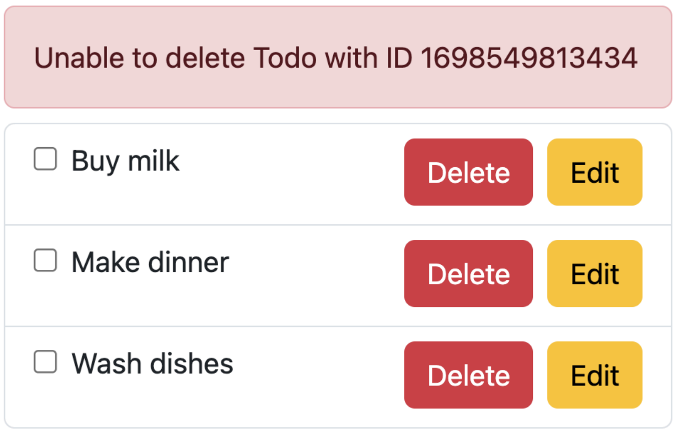
//

```
  });
  app.put("/a5/todos/:id", (req, res) => {
    const { id } = req.params;
    const todo = todos.find((t) => t.id === parseInt(id));
    if (!todo) {
      res.status(404)
        .json({ message: `Unable to update Todo with ID ${id}` });
      return;
    }
    todo.title = req.body.title;
    todo.description = req.body.description;
    todo.due = req.body.due;
    todo.completed = req.body.completed;
    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/a5/todos/:id/delete route and then try to update or delete the same item using the user interface.

src/Labs/a5/WorkingWithArrays.ts

```
function WorkingWithArrays() {
  const [errorMessage, setErrorMessage] = useState(null);
  const deleteTodo = async (todo) => {
    try {
      const response = await axios.delete(
        `${API}/${todo.id}`);
      setTodos(todos.filter((t) => t.id !== todo.id));
    } catch (error) {
      console.log(error);
      setErrorMessage(error.response.data.message);
    }
  };
  const updateTodo = async () => {
    try {
      const response = await axios.put(
        `${API}/${todo.id}`, todo);
      setTodos(todos.map((t) => (
        t.id === todo.id ? todo : t)));
    } catch (error) {
      console.log(error);
      setErrorMessage(error.response.data.message);
    }
   ...
  };
  return (
    <div>
      ...
      {errorMessage && (
        <div className="alert alert-danger mb-2 mt-2">
          {errorMessage}
        </div>
      )}
      ...
    </div>
  );
}
```


Unable to delete Todo with ID 1698549813434

4 Implementing the Kanbas Node.js HTTP Server

4.1 Create a Database

In previous assignments we created a ***Database*** component that collected all the data files into a single access point. The data should actually live on the server side and/or in an actual database. We'll first move the data to the server in this assignment, and then into a database in the next assignment. Create a folder called ***Kanbas*** at the root of your Node.js project. In the ***Kanbas*** folder, create a ***Database*** folder and copy all the JSON files from your React.js project. Then change the extensions of all the JSON files to be JavaScript files, e.g., rename ***courses.json*** to ***courses.js***. At the top of all the new JavaScript data files, add 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",
    image: "rocket-propulsion.png",
  }, ...
]
```

Do the same for all the JSON files and update the import statements in the ***Database*** component as shown below.

Kanbas/Database/index.js

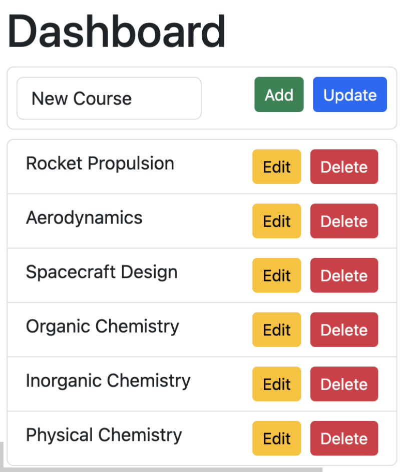
```
import courses from "./courses.js";
import modules from "./modules.js";
import assignments from "./assignments.js";
import users from "./users.js";
import grades from "./grades.js";
import enrollments from "./enrollments.js";
export default { courses, modules, assignments, users, grades, enrollments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

Feel free to use the following data, but you'll need to convert them to JavaScript files as described earlier. Here's a set of sample data files you can use.

- courses.js   modules.js   assignments.js
- grades.js   enrollment.js   users.js

4.2 Implementing Routes for Courses

4.2.1 Retrieving Courses



Now that we moved the ***Database*** to the server, we have to make it available to the React.js client application through a Web ***API*** (***Application Programming Interface***). The exercise below makes the courses available at ***http://localhost:4000/api/courses***. The screenshot here on the right is for illustration purposes only. Your actual implementation might look different.

Kanbas/courses/routes.js

```
import Database from "../Database/index.js";
export default function CourseRoutes(app) {
  app.get("/api/courses", (req, res) => {
    const courses = Database.courses;
    res.send(courses);
  });
}
```

```
//
```

In ***App.js*** import the new routes and pass a reference to the express module to the routes.

App.js

```
import express from "express";
import Lab5 from "./Lab5.js";
import CourseRoutes from "./Kanbas/courses/routes.js";
```

```
//
```

```
import cors from "cors";
const app = express();
app.use(cors());
CourseRoutes(app);
Lab5(app);
app.listen(4000);
```

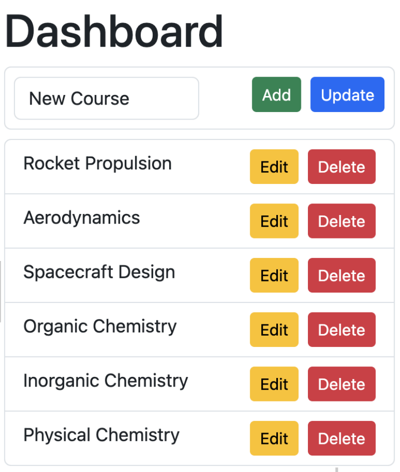
Back in the Kanbas component, a ***useEffect*** can fetch all the courses from the server on component load and update the ***courses*** state variable that populates the ***Dashboard***. Remove ***Database*** references from the user interface.

src/Kanbas/index.ts

```
~~import db from "./Database";~~
import { useState, useEffect } from "react";
import axios from "axios";
function Kanbas() {
  const [courses, setCourses] = useState<any[]>([]);
  const COURSES_API = "http://localhost:4000/api/courses";
  const findAllCourses = async () => {
    const response = await axios.get(COURSES_API);
    setCourses(response.data);
  };
  useEffect(() => {
    findAllCourses();
  }, []);
  ...
  return ( ... );
}
```

```
//
```

4.2.2 Creating New Courses



Now add 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 ***Dashboard***. The new course is given a new unique identifier and sent back to the client in the response.

Kanbas/courses/routes.js

```
import Database from "../Database/index.js";
function CourseRoutes(app) {
  app.post("/api/courses", (req, res) => {
    const course = { ...req.body,
      _id: new Date().getTime().toString() };
    Database.courses.push(course);
    res.send(course);
  });
  app.get("/api/courses", (req, res) => {...});
}
```

```
//
```

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

src/Kanbas/index.ts

```
import axios from "axios";
function Kanbas() {
  ...
  const addNewCourse = async () => {
    const response = await axios.post(COURSES_API, course);
    setCourses([ ...courses, response.data ]);
  };
  const findAllCourses = async () => {...};
```

```
  useEffect(() => {...}, []);
  return ( ... );
}
```

4.2.3 Deleting a Course

The exercise below implements a ***delete*** route that parses the ***id*** of course as a path parameter and removes the corresponding course from the ***Database***'s ***courses*** array. Status 204 is sent back as the response.

Kanbas/courses/routes.js

```
import Database from "../Database/index.js";
function CourseRoutes(app) {
  app.delete("/api/courses/:id", (req, res) => {
    const { id } = req.params;
    Database.courses = Database.courses
      .filter((c) => c._id !== id);
    res.sendStatus(204);
  });
  app.post("/api/courses", (req, res) => {...});
  app.get("/api/courses", (req, res) => {...});
}
```

```
//
```

Back in the ***Kanbas*** component, refactor the ***deleteCourse*** function so that it ***delete***s the course to 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.ts

```
import axios from "axios";
function Kanbas() {
  const deleteCourse = async (courseId: string) => {
    const response = await axios.delete(
      `${COURSES_API}/${courseId}`
    );
    setCourses(courses.filter(
      (c) => c._id !== courseId));
  };
  const addCourse = async () => {...};
  const findAllCourses = async () => {...};
  useEffect(() => {...}, []);
  return ( ... );
}
```

```
//
```

4.2.4 Updating a Course

The exercise below implements a ***put*** route that parses the ***id*** of course as a path parameter and updates the corresponding course with the updates in HTTP request body. The ***Database***'s ***courses*** array is updated with a new version of the corresponding course. Successful status 204 is sent back as the response.

Kanbas/courses/routes.js

```
import Database from "../Database/index.js";
function CourseRoutes(app) {
  app.put("/api/courses/:id", (req, res) => {
    const { id } = req.params;
    const course = req.body;
    Database.courses = Database.courses.map((c) =>
      c._id === id ? { ...c, ...course } : c
    );
    res.sendStatus(204);
  });
  app.delete("/api/courses/:id", (req, res) => {...});
```

```
//
```

```
  app.post("/api/courses", (req, res) => {...});
  app.get("/api/courses", (req, res) => {...});
}
```

Back in the ***Kanbas*** component, refactor the ***updateCourse*** function so that it ***put***s 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.ts

```
import axios from "axios";
function Kanbas() {
  const updateCourse = async () => {
    const response = await axios.put(
      `${COURSES_API}/${course._id}`,
      course
    );
    setCourses(
      courses.map((c) => {
        if (c._id === course._id) {
          return course;
        }
        return c;
      })
    );
  };
  const addCourse = async () => {...};
  useEffect(() => {...}, []);
  return ( ... );
}
```

```
// note the async keyword
```

4.2.5 Retrieve a Course by their ID

The exercise below implements a ***get*** route that parses the ***id*** of course as a path parameter and responds with the corresponding course in the ***Database***'s ***courses*** array. If the course is not a status code of 404 is sent back along with a short message in the response. If the course is found it is sent in the response.

Kanbas/courses/routes.js

```
import Database from "../Database/index.js";
function CourseRoutes(app) {
  app.get("/api/courses/:id", (req, res) => {
    const { id } = req.params;
    const course = Database.courses
      .find((c) => c._id === id);
    if (!course) {
      res.status(404).send("Course not found");
      return;
    }
    res.send(course);
  });
  app.put("/api/courses/:id", (req, res) => {...});
  app.delete("/api/courses/:id", (req, res) => {...});
}
```

```
//
```

When the user clicks on a course in the ***Dashboard***, the user interface navigates to the ***Courses*** component that displays the course selected in the ***Dashboard***. The ***Courses*** component parses the ***courseId*** from the path and used to retrieve the course from the local ***courses*** property passed in from the ***Kanbas*** component. Let's refactor the ***Courses*** component so that it retrieves the selected course from the server route implemented above. To do this, declare ***course*** state variable that we can populate with the course from the server. On load, ***useEffect*** invokes the asynchronous function ***findCourseById*** which retrieves the course from the server by it's ID and then populates the ***course*** state variable with the response. Confirm that selecting a course in the ***Dashboard*** navigates to the ***Courses*** screen and that it displays the correct course.

src/Kanbas/Courses/index.ts

```
import { useState, useEffect } from "react";
import axios from "axios";
function Courses(~~{ courses }~~) {
  const { courseId } = useParams();
  const COURSES_API = "http://localhost:4000/api/courses";
  const [course, setCourse] = useState<any>({ _id: "" });
  const findCourseById = async (courseId?: string) => {
    const response = await axios.get(
      `${COURSES_API}/${courseId}`
    );
    setCourse(response.data);
  };
  ~~const course = courses~~
    ~~.find((course) => course._id === courseId);~~
  useEffect(() => {
    findCourseById(courseId);
  }, [courseId]);
  return ( ... );
}
```

```
//
```

4.3 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.

4.3.1 Retrieving Modules for Course

Let's first implement retrieving (reading) the modules for a given course. The following route retrieves the modules for the course ID encoded in the path (cid). The course ID is parsed from the path and then used to filter the modules for that course.

Kanbas/modules/routes.js

```
import db from "../Database/index.js";
function ModuleRoutes(app) {
  app.get("/api/courses/:cid/modules", (req, res) => {
    const { cid } = req.params;
    const modules = db.modules
      .filter((m) => m.course === cid);
    res.send(modules);
  });
}
export default ModuleRoutes;
```

```
//
```

Import the new routes in ***App.js*** as shown below.

App.js

```
import express from "express";
import Lab5 from "./Lab5.js";
import CourseRoutes from "./Kanbas/Courses/routes.js";
import ModuleRoutes from "./Kanbas/Modules/routes.js";
import cors from "cors";
const app = express();
app.use(cors());
app.use(express.json());
ModuleRoutes(app);
CourseRoutes(app);
```

```
// import the new routes

// configure JSON HTTP body parsing FIRST
// and THEN configure new routes
```

```
Lab5(app);
app.listen(4000);
```

When we worked on the courses, we implemented all the ***axios*** operations within the components themselves, e.g., ***Kanbas/index.ts*** and ***Courses/index.ts***. This is, in general, a bad practice and is instead a better to implement all communication related logic in a separate file such as the ***client.js*** file below. Implement ***findModulesForCourse*** function shown below which retrieves the modules for a given course.

src/Kanbas/Courses/Modules/client.ts

```
import axios from "axios";
const COURSES_API = "http://localhost:4000/api/courses";
export const findModulesForCourse = async (courseId) => {
  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.ts

~~import db from "../Database";~~

```
const initialState = {
  modules: [],
  module: { name: "New Module 234",
            description: "New Description" },
};
const modulesSlice = createSlice({
  name: "modules",
  initialState,
  reducers: {
    setModules: (state, action) => {
      state.modules = action.payload;
    },
    addModule: (state, action) => {...},
    deleteModule: (state, action) => {...},
    updateModule: (state, action) => {...},
    setModule: (state, action) => {...},
  },
});
export const { addModule, deleteModule,
  updateModule, setModule, setModules } =
  modulesSlice.actions;
export default modulesSlice.reducer;
```

```
//
```

In the ***Modules/list.ts*** component, import the new ***setModules*** reducer function and the new ***findModulesForCourse*** service function. Use a ***useEffect*** function to invoke the service function and dispatch the modules from the service to the reducer with the ***setModules*** function as shown below. Confirm that navigating to a course populates the corresponding modules.

src/Kanbas/Courses/Modules/List.ts

```
import { useParams } from "react-router-dom";
import { useSelector, useDispatch } from "react-redux";
import React, { useEffect, useState } from "react";
import {
  addModule, deleteModule, updateModule, setModule,
  setModules,
} from "./reducer";
import { findModulesForCourse } from "./client";
```

```
//
```

```
function ModuleList() {
  const { courseId } = useParams();
  useEffect(() => {
    findModulesForCourse(courseId)
      .then((modules) =>
        dispatch(setModules(modules))
    );
  }, [courseId]);
  ...
  return (...);
}
export default ModuleList;
```

4.3.2 Creating Modules for a Course

Now let's implement the ***create*** in ***CRUD***. In the ***modules/routes.js***, implement a route that receives a new module posted from the React.js client application embedded in the Request body. Create a new module object setting its ***course*** property to the ***cid*** path parameter and push it to the ***db.modules*** array. Reply to the client with the newly created module as shown below.

Kanbas/modules/routes.js

```
import db from "../Database/index.js";
function ModuleRoutes(app) {
  app.post("/api/courses/:cid/modules", (req, res) => {
    const { cid } = req.params;
    const newModule = {
      ...req.body,
      course: cid,
      _id: new Date().getTime().toString(),
    };
    db.modules.push(newModule);
    res.send(newModule);
  });
  app.get("/api/courses/:cid/modules", (req, res) => {...});
}
export default ModuleRoutes;
```

```
//
```

Add ***createModule*** client function in ***client.js*** as shown below. It should post a new ***module*** object in the body and encode the ***courseId*** in the path. The response's ***data*** contains the newly created module from the server. We'll append the new module to the cached modules in the module reducer.

src/Kanbas/Courses/Modules/client.ts

```
import axios from "axios";
const COURSES_API = "http://localhost:4000/api/courses";
export const createModule = async (courseId, module) => {
  const response = await axios.post(
    `${COURSES_API}/${courseId}/modules`,
    module
  );
  return response.data;
};
export const findModulesForCourse = async (courseId) => {...};
```

```
//
```

In the module's ***list*** component, import the new ***createModule*** client function and invoke it with the ***courseId*** and ***module*** object. Dispatch the new ***module*** from the server to the ***addModule*** reducer function which will add it to the reducer's ***modules*** state variable. Confirm that you can create new modules for the current course.

src/Kanbas/Courses/Modules/list.ts

```
import {
  addModule,  deleteModule,  updateModule,
```

```
//
```

```
  setModule,  setModules,
} from "./reducer";
import { findModulesForCourse, createModule } from "./client";
function ModuleList() {
  ...
  const handleAddModule = () => {
    createModule(courseId, module).then((module) => {
      dispatch(addModule(module));
    });
  };
  ...
  return (
    <ul className="list-group">
      <li className="list-group-item">
        <button
          onClick={handleAddModule}>
          Add
        </button>
      </li>
    </ul>
  );
}
export default ModuleList;
```

If not already implemented, create an ***addModule*** reducer function that appends a new modules at the beginning of ***modules*** state variable as shown below.

src/Kanbas/Courses/Modules/reducer.ts

```
const initialState = {
  modules: [],
  module: { name: "New Module 234",
            description: "New Description" },
};
const modulesSlice = createSlice({
  name: "modules",
  initialState,
  reducers: {
    addModule: (state, action) => {
      state.modules = [action.payload, ...state.modules];
    },
    deleteModule: (state, action) => {...},
    updateModule: (state, action) => {...},
    setModule:    (state, action) => {...},
    setModules:   (state, action) => {...},
  },
});
export const { addModule, deleteModule,
  updateModule, setModule, setModules } =
  modulesSlice.actions;
export default modulesSlice.reducer;
```

```
//
```

4.3.3 Deleting a Module

Now let's implement the ***delete*** in ***CRUD***. In the ***modules/routes.js*** add a route that handles an ***HTTP DELETE*** method with the module's ID embedded in the URL as shown below. The ***mid*** request parameter is used to filter out the module from the ***modules*** array in the database. Respond with a 200 (or 204) status signifying success. Optionally return a 404 if the module is not found.

Kanbas/modules/routes.js

```
import db from "../Database/index.js";
function ModuleRoutes(app) {
  app.delete("/api/modules/:mid", (req, res) => {
    const { mid } = req.params;
    db.modules = db.modules.filter((m) => m._id !== mid);
    res.sendStatus(200);
```

```
//
```

```
  });
  app.post("/api/courses/:cid/modules", (req, res) => {...});
  app.get("/api/courses/:cid/modules", (req, res) => {...});
}
export default ModuleRoutes;
```

In the ***modules service*** file, implement the ***deleteModule*** service function that sends an ***HTTP DELETE*** request to the server encoding the module's ID in the path as shown below. The response's data contains a status code.

src/Kanbas/Courses/Modules/service.ts

```
import axios from "axios";
const COURSES_API = "http://localhost:4000/api/courses";
const MODULES_API = "http://localhost:4000/api/modules";
export const deleteModule = async (moduleId) => {
  const response = await axios
    .delete(`${MODULES_API}/${moduleId}`);
  return response.data;
};
export const createModule = async (courseId, module) => {...};
export const findModulesForCourse = async (courseId) => {...};
```

```
//
```

In the ***module list*** component, import the ***deleteModule*** reducer function and all the service functions as ***client***. We need to do this because some of the event handlers in the component have the same name as the service functions and so we can distinguish them by prepending the service functions with ***client***. You'll need to prepend ***client.*** to all the client functions you already used, e.g., ***findModulesForCourse*** and ***createModule*** now become ***client.findModulesForCourse***, and ***client.createModule***. Implement ***handleDeleteModule*** even handler that invokes the ***deleteModule*** service function passing it the ***moduleId*** of the module we want to remove. On response from the server, dispatch the module's ID to the reducer's ***deleteModule*** function to remove the object from the ***modules*** state variable and update the user interface.

src/Kanbas/Courses/Modules/list.ts

```
import {
  deleteModule,
  addModule,  updateModule,  setModule,  setModules,
} from "./reducer";
import * as client from "./client";
~~import { findModulesForCourse, createModule } from "./client";~~
function ModuleList() {
  const handleDeleteModule = (moduleId: string) => {
    client.deleteModule(moduleId).then((status) => {
      dispatch(deleteModule(moduleId));
    });
  };
  return (
    <ul className="list-group">
      {modules
        .filter((module) => module.course === courseId)
        .map((module, index) => (
          <li key={index} className="list-group-item">
            <button
              onClick={() => handleDeleteModule(module._id)} >
              Delete </button>
            <h3>{module.name}</h3>
          </li>
        ))}
    </ul>
  );
}
```

```
//
```

4.3.4 Update Module

Finally let's implement the ***update*** in ***CRUD***. In the ***modules routes***, implement a route that handles an ***HTTP PUT*** request with the module's ID embedded in the path and the updates in the ***HTTP body*** as shown below. The route should parse the

module's ID from the request parameters, find the module in the database, and then update the corresponding module with the values in the request body.

Kanbas/modules/routes.js

```
  ...
  app.put("/api/modules/:mid", (req, res) => {
    const { mid } = req.params;
    const moduleIndex = db.modules.findIndex(
      (m) => m._id === mid);
    db.modules[moduleIndex] = {
      ...db.modules[moduleIndex],
      ...req.body
    };
    res.sendStatus(204);
  });
  ...
```

//

In the ***service*** file in the React.js client, implement an ***updateModule*** service function that sends an ***HTTP PUT*** request to the server embedding the module's ID in the path and the updates in the ***HTTP body*** as shown below. The response's data will be a status code.

src/Kanbas/Courses/Modules/service.ts

```
...
export const updateModule = async (module) => {
  const response = await axios.
    put(`${MODULES_API}/${module._id}`, module);
  return response.data;
};
...
```

//

In the user interface, implement ***async*** function ***handleUpdateModule*** invoked when the user clicks the ***Update*** or ***Save*** button as shown below. The function should invoke the service function ***updateModule*** passing the module updates as a parameter. When the response is done, dispatch the new module to the ***updateModule*** reducer function which will replace the module in the ***modules*** state variable and update the user interface. Confirm you can update the module and that the changes persist if you refresh the user interface.

src/Kanbas/Courses/Modules/list.ts

```
  ...
  const handleUpdateModule = async () => {
    const status = await client.updateModule(module);
    dispatch(updateModule(module));
  };
  return (
    <ul className="list-group">
      <li className="list-group-item">
        <button onClick={handleAddModule}>Add</button>
        <button onClick={handleUpdateModule}>Update</button>
  ...
```

//

4.3.5 Assignments (grads only)

In your Node.js server application, implement routes for ***creating***, ***retrieving***, ***updating***, and ***deleting*** assignments. In the React.js Web application, create an ***assignment service*** 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 ***service*** 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.

5 Deploying RESTful Web service APIs on 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 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 machine, and only when we're satisfied that all works fine locally, then we can make an effort at deploying the application on remote servers.

5.1 Configuring Node applications to run in a remote server

To host our Node HTTP server on a remote server we need to make a few tweaks. For instance we can't use port 4000 like we do in our local environment. Instead we need to use the port declared in an environment variable called ***PORT*** available through ***process.env.PORT***. In your Node application, refactor ***App.js*** so that it uses the ***PORT*** environment variable if available, or uses 4000 otherwise when running locally on our machines.

App.js

```
app.listen(process.env.PORT || 4000);
```

5.2 Pushing Node server source to Github

Next let's make sure we have a proper Git repository by typing ***git init*** at the command line. It's ok if the repository was already initialized.

```
$ git  init
```

Make sure we don't include unnecessary files in the repository by listing them in ***.gitignore***. Create a new file called ***.gitignore*** if it does not already exist. Note the 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.

.gitignore

```
.idea
node_modules
.env
.env.local
.env.production
```

Use ***git add*** to add all our work 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.

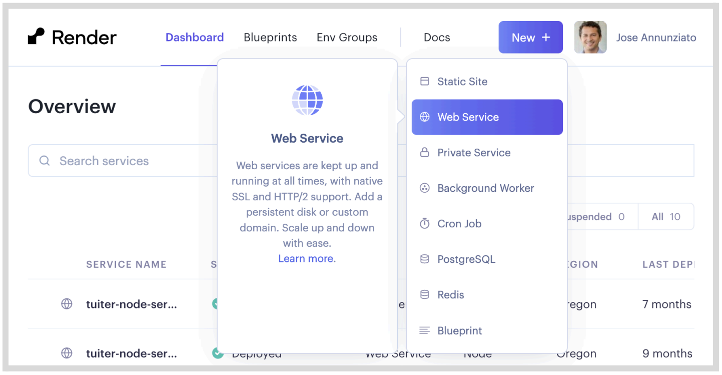
```
$ git remote add origin https://github.com/jannunzi/kanbas-node-server-app.git
```

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

```
$ git push -u origin master
```

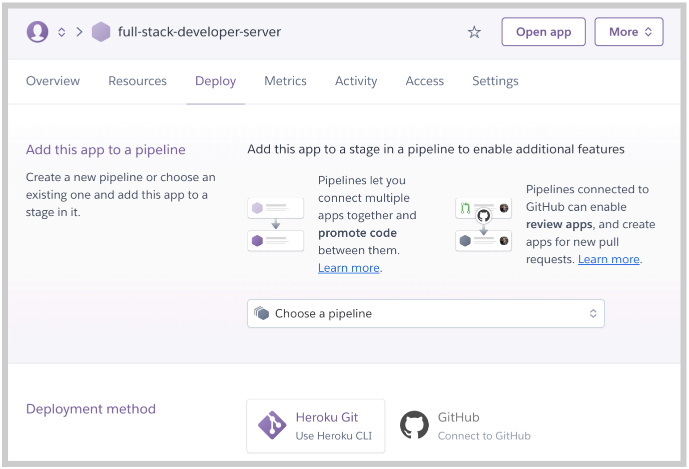
5.3 Deploying a Node server application to Render.com from Github

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.



5.4 Deploying a Node server application to Heroku from Github

If you don't already have an account at ***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.


full-stack-developer-server
Open app
More
Overview
Resources
Deploy
Metrics
Activity
Access
Settings
Add this app to a pipeline
Create a new pipeline or choose an existing one and add this app to a stage in it.
Add this app to a stage in a pipeline to enable additional features
Pipelines let you connect multiple apps together and promote code between them. Learn more.
Pipelines connected to GitHub can enable review apps, and create apps for new pull requests. Learn more.
Choose a pipeline
Deployment method
Heroku Git
Use Heroku CLI
GitHub
Connect to GitHub

5.5 Integrating Netlify React applications with remote Node servers

Let's now deploy our React client application to a remote server on Netlify. Currently our client application connects to a local Node server, which is fine when it's running and developing locally on your personal computer, but it should use the remote server deployed to Render or Heroku when it's also running remotely on Netlify. To test the remote connection works, refactor the ***Kanbas*** component so that the ***COURSE_API*** uses the remote server URL as shown below. Confirm that navigating to the ***Dashboard*** still renders the courses. There might be a delay from the response of the remote server since ***Render.com*** will spin down servers in their free tiers.

src/Kanbas/index.ts

```
function Kanbas() {
  const COURSES_API = "https://kanbas-node-server-app.onrender.com/api/courses";
  const findAllCourses = async () => {
    const response = await axios.get(COURSES_API);
    setCourses(response.data);
  };
  useEffect(() => {
    findAllCourses();
  }, []);
```

Also refactor ***Modules/client.js*** so that it uses the remote server instead.

src/Kanbas/Modules/client.js

```
import axios from "axios";
const COURSES_API = "https://kanbas-node-server-app.onrender.com/api/courses";
const MODULES_API = "https://kanbas-node-server-app.onrender.com/api/modules";
```

Reload the ***Dashboard*** to verify the courses render from the remote server. Navigate to one of the courses and then to their ***Modules*** section to confirm the ***ModuleList*** component renders the modules for the selected course.

5.6 Configuring Remote Environments

This is all and good and would work if we deploy the React application as is, but we would like the URL to point to the local Node server when developing locally, and use the remote server when deployed remotely without having to change the URLs manually ourselves. To automate which URL to use in what environment we can use environment variables to define application wide constants based on the environment the application is running in. In React projects, environment variables can be declared in an ***.env.local*** file at the root of the project as shown below.

.env.local

```
REACT_APP_API_BASE=http://localhost:4000
```

Variables must start with ***REACT_APP_*** and can be accessed from the React code using ***process.env*** as shown below.

src/Kanbas/index.ts

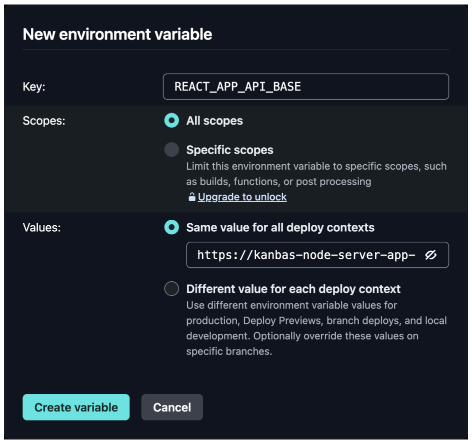
```
...
const API_BASE = process.env.REACT_APP_API_BASE;
function Kanbas() {
  const COURSES_API = `${API_BASE}/api/courses`;
  const findAllCourses = async () => {
    const response = await axios.get(COURSES_API);
    setCourses(response.data);
  };
  useEffect(() => {
    findAllCourses();
  }, []);
...
```

Refactor ***Kanbas*** component and the ***Modules/client.ts*** to use the ***REACT_APP_API_BASE*** environment variable as shown above and below. Confirm that the ***Dashboard*** and ***ModuleList*** components still render the courses and modules as expected.

src/Kanbas/Modules/client.ts

```
const API_BASE = process.env.REACT_APP_API_BASE;
const COURSES_API = `${API_BASE}/api/courses`;
const MODULES_API = `${API_BASE}/api/modules`;
```

We have successfully configured our React project to use the local Node server when it's running in our local development environment. But what about when we deploy and run the same React project remotely on Netlify? We want the React project to use the remote Node HTTP server running on Render.com or Heroku when it runs on Netlify. To accomplish this we need to configure an environment variable on Netlify that configures the ***REACT_APP_API_BASE*** to the URL of the remote Node server running on Render.com (or Heroku).


New environment variable

Key: REACT_APP_API_BASE

Scopes: [✓ All scopes] Specific scopes
Limit this environment variable to specific scopes, such as builds, functions, or post processing
Upgrade to unlock

Values: [✓ Same value for all deploy contexts]
https://kanbas-node-server-app-
Different value for each deploy context
Use different environment variable values for production, Deploy Previews, branch deploys, and local development. Optionally override these values on specific branches.

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_API_BASE*** in the ***Key*** field, and then in the ***Values*** field, copy and paste the root URL of the application running on Render.com or Heroku.com, 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 ***ModuleList*** still renders the modules for the selected course.

Now, go back to all the lab exercises and replace the use of ***localhost:4000***, with this new environment variable so that the labs also work when deployed to 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 exercises in this chapter. 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.js (or Heroku) and confirm they integrate. Submit links to both your GitHub repositories as well as the Netlify and remote Node URLs where the branches deployed. Here's an example on the steps:

Create a branch called a5

```
git  checkout  -b  a5

# do all your work
```

Once you've completed all your work, add, commit and push your work to the remote repositories.

Add, commit and push the new branch

```
git  add  .
git  commit  -am  "a5 REST fa23"
git  push
```

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

1. The new URLs where your ***a5*** branches deployed to on Netlify and Render.js (or Heroku)
2. The link to your new branches in GitHub for the React and Node.js projects