Implementing HTTP Servers and RESTful APIs with Node.js

# 1 Introduction

During the 90s, the World Wide Web adoption grew exponentially. Various commencial ventures explored a plethora of use cases that revolutionalized 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., ***tuits.json***, ***profile.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 Tuiter 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 demonstrations and exercises in previous chapters, implementing the React.js Web application. Nevertheless it's useful to review and confirm you have a working Node.js installation so using your computer terminal or console application, type the following to check the version of Node installed in your machine.

| **$ node -v**  **v16.13.2** |
| --- |

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 16.14.0 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 the 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 tuiter-node-server-app**  **$ cd tuiter-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: (tuiter-node-server-app)  version: (1.0.0)  description: Node.js HTTP Web server for the Tuiter application  entry point: (index.js)  test command:  git repository: https://github.com/jannunzi/tuiter-node-server-app  keywords: Node, REST, Server, HTTP  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 server application

Open the project created earlier with the IDE of your choice, e.g, [IntelliJ](https://www.jetbrains.com/idea/) or [Visual Studio Code](https://code.visualstudio.com/), 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 an 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*** or ***Heroku***. New libraries are installed in a new folder called ***node\_modules***. More Node.js packages can be found at [npmjs.com](https://www.npmjs.com).

The following ***app.js*** implements an HTTP server that responds ***Hello World!*** when the server receives an HTTP request at the URL [***http://localhost:4000/hello***](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***. Developers 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'**)  **const** app = *express*()  app.**get**(**'/hello'**, (req, res) => {res.**send**(**'Hello World!'**)})  app.listen(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 at ***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** |
| --- |

Now instead of using the node command to start the server, use ***nodemon*** instead 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!'**)})  app.**get**(**'/'**, (req, res) => {res.**send**(**'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": "tuiter-node-server-app",**  **...**  **"scripts": {**  **"test": "echo \"Error: no test specified\" && exit 1",**  **"start": "node app.js"**  **},** |

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); |

## 2.8 Exercises

| 1. Create the Node.js project as described in this section 2. Implement the ***hello.js*** as described in this section 3. Implement the HTTP server described in ***app.js*** | 1. Configure ***nodemon*** as described in this section 2. Configure Node.js project to use ***ES6*** as described in this section |
| --- | --- |

# 3 Creating HTTP controllers

The ***app.js*** file creates and configures an HTTP server listening for incoming HTTP requests. So far we've created a simple ***hello*** HTTP ***endpoint*** that responds with a simple string. Throughout this and later chapters we're going to create quite a few other HTTP endpoints, too many to define them all in ***app.js***. Let's move both HTTP endpoints to a separate file called ***hello-controller.js*** under a new folder called ***controllers*** as shown bellow. All subsequent controllers will be created under this ***controllers*** folder.

| ***app.js*** | ***controllers/hello-controller.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); | ***app***.get(**'/hello'**, (req, res) => {  res.send(**'Life is good!'**)  });  ***app***.get(**'/'**, (req, res) => {  res.send(**'Welcome to Full Stack Development!'**)  }); |

In the context of HTTP servers and Web APIs, ***controllers*** are functions, classes, or modules whose only role is to handle HTTP requests and participate in a ***client/server*** architecture. In our case ***hello-controller*** handles HTTP requests for a ***hello*** greeting and responds with a friendly reply. We're not done though. Notice that ***hello-controller.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.

| ***controllers/hello-controller.js*** | ***app.js*** |
| --- | --- |
| **const** *HelloController* = (app) => {  app.get(**'/hello'**, (req, res) => {  res.send(**'Life is good!'**)  })  app.get(**'/'**, (req, res) => {  res.send(**'Welcome to Full Stack Development!'**)  })  }  **export default** *HelloController*; | **import *express* from 'express'**  **import** *HelloController*  **from "./controllers/hello-controller.js"**  **const** app = *express*()  *HelloController*(app)  app.listen(4000) |

## 

## 3.1 Requesting data from a Web server with a browser

Let's practice some more creating controllers for various types of data. Let's create several HTTP endpoints to work with data related to Tuiter users. In a new folder ***controllers/users***, create a file that holds data for a couple of users as shown below. In the following sections we're going to learn how to retrieve and manipulate this data collection.

| ***controllers/users/users.js*** |
| --- |
| **export default [**  **{ "username": "alice", "type": "FACULTY", "\_id": "123" },**  **{ "username": "bob", "type": "STUDENT", "\_id": "234" },**  **{ "username": "charlie", "type": "FACULTY", "\_id": "345" }**  **]** |

Controllers are responsible for defining HTTP endpoints that client applications, such as React and mobile applications, can invoke through a request, cause some function execution on the server, and respond with a result. Creating a controller per type of data is a common strategy to break up the source code. The ***user controller*** below will implement several HTTP endpoints to create, read, update, and delete users. It is a common strategy to group together ***CRUD*** (create, read, update and delete) operations under one controller. Function ***findUsers*** below retrieves the list of all users from the server and is mapped to the HTTP endpoint ***/api/users***.

| ***controllers/users/users-controller.js*** | |
| --- | --- |
| **import** people **from './users.js'**  **let** users = people  **const** *UserController* = (app) => {  app.get(**'/api/users'**, findUsers)  }  **const** findUsers = (req, res) => {  res.json(users)  }  **export default** *UserController* | *// import the array of users*  *// use express instance app to declare HTTP GET*  *// request pattern /api/users to call a function*  *// function runs when /api/users requested*  *// responds with array of users*  *// exports so app.js can import* |

Import the user controller in the server file and pass it an instance of the express library ***app***. Restart the server and point the browser to <http://localhost:4000/api/users> and confirm that an array of users appears.

| ***app.js*** | |
| --- | --- |
| **import *express* from 'express'**  **import** *HelloController*  **from "./controllers/hello-controller.js"**  **import** *UserController*  **from "./controllers/users/users-controller.js"**  **const** app = *express*()  *HelloController*(app)  *UserController*(app)  app.listen(4000) | *// import controller*  *// pass it app* |

## 3.2 Sending query parameters to a Web server

The previous example demonstrated retrieving data from a server. We can also send data to a server as ***query string parameters***, ***path parameters***, or embedded in the ***request body***. The example below demonstrates how to work with ***query string parameters*** encoded at the end of a URL after a question mark (?). Query string parameters are name value pairs separated by ampersands (&). The example below refactors the ***findUsers*** function to check for query strings with the ***type*** of user we want, and retrieve all of them if ***type*** is omitted.

| ***controllers/users/users-controller.js*** | |
| --- | --- |
| **const** findUsers = (req, res) => {  **const** type = req.**query**.type  **if**(type) {  **const** usersOfType = users  .filter(u => u.**type** === type)  res.json(usersOfType)  **return**  }  res.json(users)  } | *// retrieve type parameter from query*  *// if type parameter in query*  *// find users of that type*  *// respond with users of that type*  *// return so it doesn't continue*  *// otherwise respond with all users* |

The query is available in the request object (***req***) under the ***query*** property implemented as a map containing each query variable as a key into the map. Try <http://localhost:4000/api/users?type=STUDENT> and confirm that the server responds with just ***bob***. Now try <http://localhost:4000/api/users?type=FACULTY> and confirm that the server responds with ***alice*** and ***charlie***. Alternatively we could have used ***req.query['type']*** since it is an equivalent syntax in JavaScript.

## 3.3 Sending path parameters to a Web server

Alternatively data can be encoded as part of the URL. The example below illustrates encoding the ID of a user as part of the URL path pattern ***/api/users/:uid***. The colon (:) followed by ***uid*** declares a placeholder that matches any literal string. The actual value in the placeholder can be retrieved using ***uid*** as a key into the request's ***params*** map. JavaScript maps can be accessed using either the ***dot notation*** syntax as shown below, or the ***square bracket*** notation, e.g., ***req.params["uid"]***.

| ***controllers/users/users-controller.js*** | |
| --- | --- |
| **const** *UserController* = (app) => {  app.get(**'/api/users'**, findUsers);  app.get(**'/api/users/:uid'**, findUserById);  }  **const** findUserById = (req, res) => {  **const** userId = req.**params**.uid;  **const** user = users  .find(u => u.**\_id** === userId);  res.json(user);  } | *// map path pattern to handler function*  *// function called if URL matches pattern*  *// get uid from request parameter map*  *// find user in users array whose \_id*  *// matches userId retrieved from params*  *// respond to client with user found* |

Try <http://localhost:4000/api/users/123> and confirm that the server responds with a single object for ***alice***. Note it's not an array.

## 3.4 Exercises

1. Implement ***users-controller.js*** as described in this section
2. Implement ***findUsers*** as described in this section
3. Implement ***findUserById*** as described in this section

# 4 Interacting with a Web server using Postman

So far we've interacted with the Web server by just typing URLs into a browser and viewing the servers response in the browser's window. Unfortunately this takes us only so far since we can only generate ***GET*** requests from a browser's URL field. To generate other types of HTTP requests such as ***POST***, ***PUT***, and ***DELETE***, we're going to need either JavaScript or tools such as ***Postman***, a popular networking test tool well worth learning about.

## 4.1 Installing and running Postman

Postman is a tool for creating, scripting and testing HTTP requests. To install postman, head over to their Website at [***postman.com***](http://postman.com) and download the latest version of the tool. Run the installer and follow the instructions accepting all the default configurations. Once installed, run postman to start using the tool. To practice, reimplement the earlier queries using Postman.

## 4.2 Requesting data from a Web server using Postman

Postman ***collections*** help organize related sets of HTTP requests into groups. Create a collection to contain all the requests we're going to practice with in this section. On the far left top of the main window, select ***Collections*** and click the ***plus*** sign to create a new collection named ***cs-4550***, ***cs-5610***, ***web-dev***, or something memorable. Click the ***star*** next to the new collection to favorite it and have it listed near the top of the collections list so you can find it later. While the collection is selected, click the three horizontal dots and select ***Add request*** and name the new request ***find all users*** in the window that appears on the right. Under the name of the request, make sure the HTTP method is the default ***GET***. In the field ***Enter request URL***, type the URL used earlier to retrieve all the users, e.g., <http://localhost:4000/api/users>, press the blue ***Send*** button on the right and confirm that the same array of users appears below in the ***Response*** window.

## 4.3 Posting data to a Web server using Postman

In prior exercises we sent data to the server by encoding it as part of the path or query parameters which is limited and unsafe since browsers impose a limit in the length of the URL and it is sent to the server in plain text, visible to any prying eyes between the client and the server. Alternatively data can be sent to the server embedded in the ***body*** of the HTTP request where it can be encrypted for safe transmission. The function ***createUser*** below can read data posted to the server, embedded in the HTTP request body, and interpret it as a ***new user*** and store it in the users array.

| ***controllers/users/users-controller.js*** | |
| --- | --- |
| **const** *UserController* = (app) => {  app.get(**'/api/users'**, findUsers);  app.get(**'/api/users/:uid'**, findUserById);  app.post(**'/api/users'**, createUser);  }  **const** createUser = (req, res) => {  **const** newUser = req.**body**;  newUser.**\_id** = (**new *Date***()).getTime() + **''**;  users.push(newUser);  res.json(newUser);  } | *// map URL pattern to handler function*  *// function invoked if URL matches pattern*  *// extract new user from BODY in request*  *// add an \_id property with unique timestamp*  *// append new user to users array*  *// respond with new user to client* |

Out of the box express does not know how to extract data from an HTTP body. Express defines a JSON middleware to parse data from the body that can be registered as a middleware. All requests will first go through this middleware parsing the HTTP body into a JSON object added to the request object in a new ***body*** property that later HTTP handlers can access.

| ***app.js*** | |
| --- | --- |
| **import *express* from 'express'**;  **import** *UserController*  **from "./controllers/users-controller.js"**;  **const** app = *express*();  app.**use**(*express*.***json***());  *HelloController(app);*  *UserController*(app);  app.listen(4000); | *// parse JSON from HTTP request body* |

To test the new endpoint, create a ***POST*** request in Postman that sends a user object embedded in its body. In the same collection used earlier, create another HTTP request called ***create user*** in the window that appears on the right. Under the name of the request, make sure to select ***POST*** as the HTTP method instead of the default ***GET***. In the field ***Enter request URL***, type the same URL used earlier to retrieve all the users, e.g., [http:// localhost:4000/api/users](http://localhost:4000/api/users). Under the URL select the ***Body*** tab, then ***raw***, and then ***JSON*** from the ***Text*** dropdown. In the textarea that appears under the radio buttons, type a JSON object representing a new user object as shown below. Note we are not including a ***\_id*** property since the ***createUser*** HTTP handler will create that for us.

| {  "username": "dan",  "type": "STUDENT"  } |
| --- |

Press the blue ***Send*** button on the right and confirm that the response contains the same object posted, but with an added ***\_id*** property with some random numeric string. Use the ***find all users*** request and confirm that the new user was added to the array of users.

## 4.4 Deleting data from a Web server using Postman

We can now create and retrieve data in a server, two of the ***four basic operations***: ***create***, ***read***, ***update***, and ***delete***. Let's now learn how to remove data from a server with an ***HTTP delete***. In the users controller let's implement ***deleteUser*** to remove a user by their ID as shown below.

| ***users-controller.js*** |  |
| --- | --- |
| **const** deleteUser = (req, res) => {  **const** userId = req.**params**[**'uid'**];  users = users.filter(usr =>  usr.**\_id** !== userId);  res.sendStatus(200);  } | *// get user ID from path parameter uid*  *// filter out the user*  *// whose ID is the ID of the user we want to remove*  *// respond with success code* |

We'll embed the ID of the user as part of the URL. Similar to the endpoint for ***findUserById***, ***deleteUser*** encodes the ID of the user at the end of the URL. When referring to a particular instance in a collection of resources, it is a common practice to encode the ID after the plural noun referring to the collection of resources. The ***app.delete()*** below implements an HTTP handler that will delete a user when an HTTP DELETE is requested referring to a particular user.

| ***users-controller.js*** |  |
| --- | --- |
| **const** *UserController* = (app) => {  app.get(**'/api/users'**, findUsers);  app.get(**'/api/users/:uid'**, findUserById);  app.post(**'/api/users'**, createUser);  app.delete(**'/api/users/:uid'**, deleteUser);  } | *// map URL pattern to handler function* |

To test the new endpoint, create a DELETE request in Postman that sends the ID of a user we want to remove. In the same collection used earlier, create another HTTP request called ***delete user*** in the window that appears on the right. Under the name of the request, make sure to select ***DELETE*** as the HTTP method instead of the default ***GET***. In the field ***Enter request URL***, type the same URL used earlier to retrieve all the users, but this time append the ID of one of the users, e.g., let's remove bob, <http://localhost:4000/api/users/234>. Press the blue ***Send*** button on the right and confirm that the response is ***OK***. Use the ***find all users*** request and confirm that ***bob*** is no longer in the array of users.

## 4.5 Updating data in a Web server with Postman

We're done with ***create***, ***read***, and ***delete***. We just have one more to go: ***update***. For each of the operations created so far we used the following HTTP methods:

* POST to create data
* GET to retrieve (or read) data
* DELETE to remove data

It is common to use HTTP PUT for updating data. To update the collection of users we'll re-create a brand new array of users merging the old user with the new version of the user. We'll iterate through the original array of users and when we find the user that needs to be updated we'll merge the old version overriding the fields passed in the HTTP BODY as implemented in ***updateUser*** below.

| ***users-controller.js*** |  |
| --- | --- |
| **const** updateUser = (req, res) => {  **const** userId = req.**params**[**'uid'**];  **const** updates = req.**body**;  users = users.map((usr) =>  usr.**\_id** === userId ?  {...usr, ...updates} :  usr  );  res.sendStatus(200);  } | *// handle PUT /api/users/:uid*  *// get user ID from path*  *// BODY includes updated fields*  *// create a new array of users*  *// if current user's ID matches ID we want to update*  *// merge old usr with new updates*  *// otherwise keep the old user*  *// return OK* |

The ***updateUser*** function will execute when the server receives an HTTP PUT request mapped to ***/api/users/:uid*** where ***uid*** is the ID of the user being updated. The new updated user is embedded in the HTTP request body.

| ***users-controller.js*** |  |
| --- | --- |
| **const** *UserController* = (app) => {  app.get(**'/api/users'**, findUsers);  app.get(**'/api/users/:uid'**, findUserById);  app.post(**'/api/users'**, createUser);  app.delete(**'/api/users/:uid'**, deleteUser);  app.put(**'/api/users/:uid'**, updateUser);  } |  |

To test the new endpoint, create a PUT request in Postman that sends the ID of a user we want to update and the updated user encoded in the body of the message. In the same collection used earlier, create another HTTP request called ***update user*** in the window that appears on the right. Under the name of the request, make sure to select ***PUT*** as the HTTP method instead of the default ***GET***. In the field ***Enter request URL***, type the same URL used earlier to delete bob, but this time use charlie's ID, e.g., let's update charlie, <http://localhost:4000/api/users/345>. Under the URL select the ***Body*** tab, then ***raw***, and then ***JSON*** from the ***Text*** dropdown. In the textarea that appears under the radio buttons, type a JSON object representing the updated user object as shown below.

| {  "username": "charlie",  "type": "FACULTY",  "tenured": true,  "office": "LV426",  "\_id": "345"  } |  |
| --- | --- |

Press the blue ***Send*** button on the right and confirm that the response is OK. Use the ***find all users*** request and confirm that ***charlie*** is now tenured and has an office.

## 4.6 Exercises

1. Implement ***createUser*** as described in this section
2. Implement ***deleteUser*** as described in this section
3. Implement ***updateUser*** as described in this section

# 5 Implementing Tuiter RESTful Web service APIs

The URLs, operations, and formats used in the previous section collectively follow a popular conventioned called ***Representational State Tranfer*** or ***REST***. The conventions are summarized as follow.

* Use plural nouns to refer to resources
* Use POST to create new instances of those resources
* Use GET to read/retrieve existing instances of those resources
* Use PUT to update existing instances of resources
* Use DELETE to remove/delete existing instances of resources
* Encode the ID of the resource following the name of the resource

The collection of HTTP handlers that create, read, update, and delete instances of a particular resource, that collectively implement a Web API that follow the REST conventions are referred to as RESTful Web service APIs. Let's create another example so we can become familiar with the conventions and principles. In the previous section we were manipulating instances of users, and in this section we'll practice working with tuits. Let's start with a collection of tuits as the array shown below. Create the tuits in ***controllers/tuits/tuits.js***

| ***controllers/tuits/tuits.js*** |  |
| --- | --- |
| **export default [**  **{**  **"\_id": 123, "topic": "Space", "username": "SpaceX",**  **"handle": "@spacex", "time": "2h", "image": "spacex.png",**  **"title": "SpaceX's Mission",**  **"tuit": "You want to wake up in the morning and think the future is going to be great - and that’s what being a spacefaring civilization is all about. It’s about believing in the future and thinking that the future will be better than the past. And I can’t think of anything more exciting than going out there and being among the stars",**  **"liked": true, "likes": 2345,**  **"replies": 123, "retuits": 432**  **},**  **{**  **"\_id": 234, "topic": "Traffic", "username": "The Boring Company",**  **"handle": "@boringcompany", "time": "2h", "image": "boring2.jpg",**  **"title": "100s of SpaceX Starships land on Mars after a 6 month journey. 1000s of Martian colonists being building Mars Base 1",**  **"tuit": "The Boring Company (TBC) is an American infrastructure and tunnel construction services company founded by Elon Musk. Its ongoing and proposed projects are designed for intra-city ('loop') transit systems.",**  **"liked": true, "likes": 2345,**  **"replies": 123, "retuits": 432**  **},**  **];** | |

Under the ***controllers/tuits*** directory, let's implement ***tuits controller*** shown below defining the four CRUD operations create, read, update, and delete. The tuits are imported as ***posts*** and then assigned to the local array variable ***tuits***. The HTTP handlers will update the ***tuits*** array as requests come in.

| ***controllers/tuits/tuits-controller.js*** |  |
| --- | --- |
| **import posts from "./tuits.js";**  **let tuits = posts;**  **const** createTuit = (req, res) => {}  **const** findTuits = (req, res) => {}  **const** updateTuit = (req, res) => {}  **const** deleteTuit = (req, res) => {}  **export default** (app) => {  app.post(**'/api/tuits'**, createTuit);  app.get(**'/api/tuits'**, findTuits);  app.**put**(**'/api/tuits/:tid'**, updateTuit);  app.delete(**'/api/tuits/:tid'**, deleteTuit);  } | *// implement this*  *// implement this*  *// implement this*  *// implement this* |

In ***app.js***, import and initialize the new ***tuits controller*** as shown below. In the next sections we'll implement each endpoint.

| ***app.js*** |  |
| --- | --- |
| **import *express* from 'express';**  **import *HelloController***  **from "./controllers/hello-controller.js";**  **import *UserController***  **from "./controllers/users/user-controller.js";**  **import** *TuitsController*  **from "./controllers/tuits/tuits-controller.js"**;  **const** app = *express*();  app.**use**(*express*.***json***());  *TuitsController*(app);  *HelloController*(app);  *UserController*(app);  app.listen(4000); |  |

## 5.1 Retrieving data from a RESTful Web service API

Let's start with implementing the simple operation of retrieving all tuits from the server as shown below.

| ***tuits-controller.js*** |  |
| --- | --- |
| **const** findTuits = (req, res) =>  res.json(tuits); |  |

To test the new endpoint, create a GET request in Postman called ***find all tuits***. Make sure the HTTP method is the default ***GET***. In the field ***Enter request URL***, type the URL used earlier to retrieve all the users, e.g., <http://localhost:4000/api/tuits>, press the blue ***Send*** button on the right and confirm that the you get the tuits in ***tuits.js*** in the ***Response*** window.

## 5.2 Posting data to a RESTful Web service API

Let's now practice sending data embedded in the HTTP body. The function ***createTuit*** below can read data posted to the server, embedded in the HTTP request body, and interpret it as a ***new tuit*** and store in the tuits array. A new ***\_id*** value is created to uniquely identify the object.

| ***tuits-controller.js*** |  |
| --- | --- |
| **const** createTuit = (req, res) => {  **const** newTuit = req.**body**;  newTuit.**\_id** = (**new *Date***()).getTime()+**''**;  newTuit.**likes** = 0;  newTuit.**liked** = **false**;  tuits.push(newTuit);  res.json(newTuit);  } | *// retrieve data from HTTP body*  *// add \_id field as a time stamp*  *// initialize likes counter*  *// initialize liked flag*  *// append new tuit to tuits array*  *// respond with new tuit*  *// next chapter will store in database instead* |

To test the new endpoint, create a POST request in Postman that sends a tuit object embedded in its body. In the same collection used earlier, create another HTTP request called ***create tuit*** in the window that appears on the right. Under the name of the request, make sure to select ***POST*** as the HTTP method instead of the default ***GET***. In the field ***Enter request URL***, type the same URL used earlier to retrieve all the tuits, e.g., [http:// localhost:4000/api/tuits](http://localhost:4000/api/tuits). Under the URL select the ***Body*** tab, then ***raw***, and then ***JSON*** from the ***Text*** dropdown. In the textarea that appears under the radio buttons, type a JSON object representing a new tuit object as shown below. Note we are not including a ***\_id*** property since the ***createTuit*** HTTP handler will create that for us.

| {  "tuit": "Cybertrucks driving up Olympus Mons"  } |
| --- |

Press the blue ***Send*** button on the right and confirm that the response contains the same object posted, but with added ***\_id***, ***liked***, and ***likes*** properties. Use the ***find all tuits*** request and confirm that the new tuit was added to the array of tuits. Post an additional tuit, but with different content, and verify it was added as well.

| {  "tuit": "100s of SpaceX Starships land on Mars with 1000s of colonists"  } |
| --- |

## 5.3 Deleting data from a RESTful Web service API

Let's now learn how to remove data from a RESTful Web service API with an HTTP delete. In the tuits controller let's implement ***deleteTuit*** to remove a tuit by its ID encoded at the end of the URL pattern.

| ***tuits-controller.js*** |  |
| --- | --- |
| **const** deleteTuit = (req, res) => {  **const** tuitdIdToDelete = req.**params**.tid;  tuits = tuits.filter((t) =>  t.**\_id** !== tuitdIdToDelete);  res.sendStatus(200);  } | *// retrieve the ID of the tuit we want to remove*  *// filter out the tuit from the tuits array*  *// respond with success* |

To test the new endpoint, create a DELETE request in Postman that sends the ID of a tuit we want to remove. In the same collection used earlier, create another HTTP request called ***delete tuit*** in the window that appears on the right. Under the name of the request, make sure to select ***DELETE*** as the HTTP method instead of the default ***GET***. In the field ***Enter request URL***, type the same URL used earlier to retrieve all the tuits, but this time append the ID of one of the tuits, e.g., let's remove the first one, <http://localhost:4000/api/tuits/123>. Press the blue ***Send*** button on the right and confirm that the response is OK. Use the ***find all tuits*** request and confirm that the tuit is no longer in the array of tuits.

## 5.4 Updating data in a RESTful Web service API

To update a tuit we're going to find the original tuit in the collection of tuits and then merge it with the tuit updates as shown in ***updateTuit*** below.

| ***tuits-controller.js*** |  |
| --- | --- |
| **const** updateTuit = (req, res) => {  **const** tuitdIdToUpdate = req.**params**.tid;  **const** updates = req.**body**;  **const** tuitIndex = tuits.findIndex(  (t) => t.**\_id** === tuitdIdToUpdate)  tuits[tuitIndex] =  {...tuits[tuitIndex], ...updates};  res.sendStatus(200);  } | *// get ID of tuit to update from path*  *// get updates from HTTP body*  *// find index of tuit to update*  *// in the tuits array*  *// update the element in tuits array*  *// merging/updating old tuit with updates*  *// respond with success*  *// next chapter will remove from database intead* |

To test the new endpoint, create a PUT request in Postman that sends the ID of a tuit we want to update and the updated tuit encoded in the body of the message. In the same collection used earlier, create another HTTP request called ***update tuit*** in the window that appears on the right. Under the name of the request, make sure to select ***PUT*** as the HTTP method instead of the default ***GET***. In the field ***Enter request URL***, type the same URL used earlier to delete the 1st tuit, but this time use the ID of the second tuit, e.g., [http://localhost:4000/ api/tuits/234](http://localhost:4000/api/tuits/234). Under the URL select the ***Body*** tab, then ***raw***, and then ***JSON*** from the ***Text*** dropdown. In the textarea that appears under the radio buttons, type a JSON object representing the updated tuit object as shown below.

| {  "tuit": "Cybertrucks driving across Valles Marineris",  "liked": true,  "likes": 12  } |
| --- |

Press the blue ***Send*** button on the right and confirm that the response is OK. Use the ***find all tuits*** request and confirm that the tuit was updated.

## 5.5 Exercises

1. Implement ***tuiter-controller.js*** as described in this section
2. Implement ***createTuit*** as described in this section
3. Implement ***​​findTuits*** as described in this section
4. Implement ***updateTuit*** as described in this section
5. Implement ***deleteTuit*** as described in this section

# 6 Integrating React applications with RESTful Web service APIs

Now that we have a middletier, let's take a look at how to integrate our React application with the RESTful Web service APIs implemented so far. Locally the server application runs on ***localhost:4000*** and the React application runs on ***localhost:3000*** and integrating these two applications consists of exchanging HTTP messages. Browsers enforce security policies that don't allow JavaScript downloaded from one domain to communicate with other domains. Servers can be configured to allow sharing resources across domains. Let's install the ***cors*** library to configure servers to allow interoperability scripts from another domain.

| **$ npm install cors** |
| --- |

***CORS*** stands for ***Cross Origin Resource Sharing*** and establishes the rules by which resources can be shared across domains (origins). Configure CORS in app.js by importing it and using it as the first middleware.

| ***app.js*** |  |
| --- | --- |
| **import *express* from 'express'**  **import *cors* from 'cors'**  **const** app = *express*()  app.**use**(*cors*())  app.**use**(*express*.***json***()) | *// import the new cors library*  *// configure cors right after instantiating express* |

In the React application install ***axios***, a library to programmatically send and receive HTTP requests..

| **$ npm install axios** |
| --- |

In the Node.js server application, we implemented all the tuits related endpoints in one file: ***tuits-controller.js***. We'll follow a similar strategy on the React.js client application implementing all tuits related HTTP communication in ***tuits-service.js*** under a new ***services*** folder. Declare the four common CRUD operations and implement them in the following sections. The functions are all implemented as ***asynchronous*** functions that will not block the browser's sole JavaScript thread. Instead they will rely on the browser's multithreaded capabilities to send HTTP requests asynchronous and notify our functions when responses eventually resolve. We'll implement each service function in the sections that follow.

| ***src/services/tuits-service.js*** |  |
| --- | --- |
| **import *axios* from 'axios'**;  **const** TUITS\_API = **'http://localhost:4000/api/tuits'**;  **export const** *createTuit* = **async** (tuit) => {}  **export const** *findTuits* = **async** () => {}  **export const** *deleteTuit* = **async** (tuit) => {}  **export const** *updateTuit* = **async** (tuit) => {} | *// import axios*  *// location of HTTP services*  *// implement this in following sections*  *// implement this in following sections*  *// implement this in following sections*  *// implement this in following sections* |

## 6.1 Requesting data from a RESTful Web Server API from React

Let's first implement ***findTuits***, the easiest of the services that retrieves all the tuits from the server. The ***findTuits*** function sends an HTTP GET request to ***TUITS\_API*** using the ***axios.get()*** function. The request is asynchronous as evidenced by the ***async*** and ***await*** keywords and the ***response*** will be set when the request resolves from the server. The data in the response is the tuits array sent back from ***findTuits*** in ***tuits-controller.js*** and it's embedded in the response's ***data*** property.

| ***src/services/tuits-service.js*** |  |
| --- | --- |
| **export const** *findTuits* = **async** () => {  **const** response = **await *axios***.get(TUITS\_API);  **const** tuits = response.**data**;  **return** tuits;  } | *// async tags this function as asynchronous*  *// send HTTP GET request to TUITS\_API*  *// extract JSON from response from server*  *// return tuits* |

In previous assignments, reducers were used to keep track of the tuits, initialized from JSON files imported from within the reducers, but now we want to use data from the server instead. Previous implementations used reducer functions to update the state of the tuits array by adding new tuits to the state, deleting tuits, and modifying tuits in a redux store. Now we need to replace this implementation with the HTTP services we implemented in the Node.js server in the previous section. The reducers we've already implemented that create, delete, and update tuits, are all synchronous, manipulating data local to the React.js application, but we need to instead interact asynchronously with an HTTP server, while still maintaining a state with Redux. The Redux ***createAsyncThunk*** function can wrap an asynchronous HTTP function so that it can interact with a Redux reducer to store data from a remote server into a local redux store. In ***tuits-thunks.js***, create the ***findTuitsThunk*** function that wraps the ***findTuits*** HTTP service function as shown below. We'll implement thunks for each service function in later sections.

| ***src/services/tuits-thunks.js*** | |  |
| --- | --- | --- |
| **import** {*createAsyncThunk*}  **from "@reduxjs/toolkit"**  **import** \* **as** service  **from "./tuits-service"**  **export const *findTuitsThunk*** = *createAsyncThunk*(  **'tuits/findTuits'**, **async** () =>  **await** service.*findTuits*()  ) | | *// import createAsyncTrunk*  *// import all exported functions as service*  *// create thunk for findTuits*  *// give unique name, thunk invokes*  *// service function. Returned data goes in*  *// redux action's payload* |

The ***TuitList*** React component implemented in an earlier assignment can use the thunks to retrieve the tuits when the component loads. The ***useEffect*** hook declares a function callback to be invoked when the component first loads. In our case we'll dispatch the ***findTuitsThink*** that uses the service's ***findTuits*** function to retrieve all the tuits and then send to a reducer to store the new state.

| ***tuit-list.js*** |  |
| --- | --- |
| **import *React***, {*useEffect*} **from "react"**;  **import** {*useDispatch*, *useSelector*}  **from "react-redux"**;  **import** *TuitItem* **from "./tuit-item"**;  **import** {***findTuitsThunk***}  **from "../../services/tuits-thunks"**;  **const** *TuitsList* = () => {  **const** {tuits, loading} = *useSelector*(  state => state.**tuitsData**)  **const** dispatch = *useDispatch*();  *useEffect*(() => {  dispatch(***findTuitsThunk***())  }, [])  **return**(  <**ul className="list-group"**>  {  loading &&  <**li className="list-group-item"**>  Loading...  </**li**>  }  ...  </**ul**>  );  };  **export default** *TuitsList*; | *// import the thunk*  *// grab tuits and loading flag from reducer*  *// on component load*  *// invoke find tuits thunk to fetch tuits and*  *// put them in the reducer's store so we can*  *// extract them with useSelector() and render*  *// the tuits here*  *// if loading flag is true, then show a*  *// loading message while data is still*  *// coming back from the server*  *// removed for brevity* |

The reducer updates the state by setting it to the tuits from the server passed as ***payload*** in the action object. The new state is stored in the reducer's store, provided by the Provider, retrieved in ***TuitList*** with ***useSelector()*** and the component renders the array of tuits from the server. Each thunk has various states that track their lifecycle while requests are in flight, such as: ***pending***, ***fulfilled***, and ***rejected***. When the request is first sent to the server, thunks are in a ***pending*** state. When the server finally responds, thunks are in a ***fulfilled*** state. If the server times out or responds with an error, thunks are in a ***rejected*** state. Each of the states can be handled as separate reducers as shown below. Only the ***fulfilled*** state is required, but we provide examples for all three.

| ***tuits-reducer.js*** |  |
| --- | --- |
| **import** {*createSlice*}  **from "@reduxjs/toolkit"**;  **import** tuits **from './tuits.json'**;  **import** {***findTuitsThunk***}  **from "../../services/tuits-thunks"**;  **const** initialState = {  **tuits**: [],  **loading**: **false**  }  **const** tuitsSlice = *createSlice*({  **name**: **'tuits'**,  initialState,  **extraReducers**: {  [***findTuitsThunk***.**pending**]:  (state) => {  state.**loading** = **true**  state.**tuits** = []  },  [***findTuitsThunk***.**fulfilled**]:  (state, { payload }) => {  state.**loading** = **false**  state.**tuits** = payload  },  [***findTuitsThunk***.**rejected**]:  (state) => {  state.**loading** = **false**  }  },  **reducers**: { ... }  }); | *// import the thunks*  *// initial state has*  *// no tuits*  *// loading flag to display spinner*  *// same as "initialState": initialState*  *// define asynchronous reducers*  *// if request is not yet fulfilled …*  *// set loading true so UI can display spinner*  *// empty tuits since we are still fetching them*  *// when we get response, request is fulfilled*  *// we extract/destruct payload from action object*  *// turn off loading flag since we have the data*  *// payload has tuits from server and update redux state*  *// if request times out, or responds with error*  *// reset loading flag, maybe use another flag to report*  *// error*  *// we're not going to use the old reducers anymore* |

The state in the previous implementation of tuits reducer contained the array of tuits. In our new implementation the state is an object that holds the tuits in a new ***tuits*** property and another property ***loading*** to display a message while data is still loading. Let's update the store configuration to reflect this. Open the ***Tuiter*** application and confirm the tuits from the server render properly.

| ***src/tuiter/index.js*** | | |
| --- | --- | --- |
| **...**  **const** store = *configureStore*({  **reducer**: {  **who**: whoReducer,  **tuitsData**: tuitsReducer}});  **function** *Tuiter*() { ... } | | *// rename reducer property since now it's not just an array* |

## 6.2 Deleting data from a RESTful Web server API from Reac

Let's now implement the second easiest service ***deleteTuit*** implemented below. The only additional complexity is that it needs to append the ID of the tuit being removed as expected by the corresponding endpoint implemented in the ***tuits-controller.js***. The ID is appended from the tuit parameter and the response is just a status.

| ***tuits-service.js*** |  |
| --- | --- |
| **export const** *deleteTuit* = **async** (tid) => {  **const** response = **await *axios***  .delete(**`**${TUITS\_API}**/**${tid}**`**)  **return** response.**data**  } | *// send HTTP DELETE request to server*  *// append tuit's ID to URL*  *// data contains response's status we'll ignore*  *// for now* |

Implement a thunk that wraps the ***deleteTuit*** service function and puts the tuits in the action's payload. This will be useful to splice that tuit from the reducer's state's ***tuits*** property.

| ***tuits-thunks.js*** |  |
| --- | --- |
| **export const *deleteTuitThunk*** = *createAsyncThunk*(  **'tuits/deleteTuit'**,  **async** (tuitId) => {  **await** service.*deleteTuit*(tuitId)  **return** tuitId  }) | *// unique thunk identifier*  *// wraps*  *// service method*  *// return tuit ID so we can remove tuit*  *// from reducer's store* |

The ***TuitItem*** can invoke the ***deleteTuitThunk*** from the remove icon embedded in each of the tuits. It does need to pass a reference to ***dispatch*** as well as the tuit being removed.

| ***tuit-item.js*** |  |
| --- | --- |
| **~~import~~** ~~{~~*~~deleteTuit~~*~~}~~ **~~from "./tuits-reducer"~~**~~;~~  **import** {***deleteTuitThunk***} **from "../../services/tuits-thunks"**;  **const** *TuitItem* = () => {  **const** dispatch = *useDispatch*();  **const** deleteTuitHandler = (id) => {  dispatch(***deleteTuitThunk***(id));  }  **return**( ... )  }  **export default** *TuitItem*; |  |

The ***deleteTuitThink*** invokes the ***deleteTuit*** service and ***await***s the response which we know is just status information from the corresponding endpoint on the server. If the status is successful then we notify the reducer passing it the tuit being removed. The reducer creates a new state filtering out the removed tuit from the current state of tuits. We're ignoring the ***pending*** and ***rejected*** thunk states.

| ***tuits-reducer.js*** |  |
| --- | --- |
| **import** {***deleteTuitThunk***, ***findTuitsThunk***}  **from "../../services/tuits-thunks"**;  **const** tuitsSlice = *createSlice*({  **name**: **'tuits'**,  initialState,  **extraReducers**: {  // ...  [***deleteTuitThunk***.**fulfilled**] :  (state, { payload }) => {  state.**loading** = **false**  state.**tuits** = state.**tuits**  .filter(t => t.**\_id** !== payload)  },  },  **reducers**: {  //...  }  }) | *// import the new thunk*  *// handle successful response*  *// server response successful*  *// payload from action contains tuit ID to remove*  *// turn off loading flag*  *// filter out tuit whose ID matches tuit to remove*  *// we're ignoring pending and rejected thunks* |

## 6.3 Posting data to a RESTful Web server API from React

Let's now implement posting new tuits to the server in ***createTuit*** below. The ***axis.post()*** function embeds the tuit parameter in the ***body*** of the HTTP request and then ***POST***s it to ***TUITS\_API***. The response data contains the new tuit including a unique ID and initial default values such as ***likes*** set to 0 and ***liked*** set to false.

| ***tuits-service.js*** |  |
| --- | --- |
| **export const** *createTuit* = **async** (tuit) => {  **const** response = **await *axios***.post(TUITS\_API, tuit)  **return** response.**data**;  } |  |

In ***WhatsHappening***, the user interface declares a local state variable ***whatsHappening*** where we can type the new tuit from a ***textarea*** widget and then send the ***newTuit*** to the ***createTuitThunk*** along with a reference of ***dispatch***.

| ***whats-happening.js*** |  |
| --- | --- |
| **~~import~~** ~~{~~*~~createTuit~~*~~}~~  **~~from "../tuits/tuits-reducer"~~**~~;~~  **import** {***createTuitThunk***}  **from "../../services/tuits-thunks"**;  ...  **const** dispatch = *useDispatch*();  **const** tuitClickHandler = () => {  **const** newTuit = {  tuit: whatsHappening  }  dispatch(***createTuitThunk***(newTuit));  }  ... | *// wont be using the reducer function anymore*  *// we'll be using the*  *// createTuitThunk instead*  *// use thunk instead of reducer function* |

The ***createTuitThunk*** thunk posts the new ***tuit*** to the server using the corresponding service ***createTuit*** and the new tuit is dispatched to the reducer to be added to the state. The reducer updates the state appending the new tuit to the current state which makes its way through the store, provider, selector and back into the user interface for rendering. We're ignoring the ***pending*** and ***rejected*** thunks.

| ***tuits-reducer.js*** |  |
| --- | --- |
| **import** {***createTuitThunk***, ***...***}  **from "../../services/tuits-thunks"**;  ...  **const** tuitsSlice = *createSlice*({  **name**: **'tuits'**,  initialState,  **extraReducers**: {  [***findTuitsThunk***.**fulfilled**]:  (state, { payload }) => { ... },  [***deleteTuitThunk***.**fulfilled**]:  (state, { payload }) => { ... },  [***createTuitThunk***.**fulfilled**]:  (state, { payload }) => {  state.**loading** = **false**  state.**tuits**.push(payload)  },  },  });  ... | *// import the thunk*  *// when server responds*  *// payload contains new tuit*  *// clear loading flag*  *// appendnew tuit to tuits array*  *// we're ignoring pending and rejected thunks* |

## 6.4 Updating data in a RESTful Web server API from React

Finally let's implement the last of the four operations. In the ***updateTuit*** function below the ***axios.put()*** function embeds the updated tuit parameter in the ***body*** of the HTTP request and then ***PUT***s it to ***TUITS\_API*** with the tuit's ID appended at the end.

| ***tuits-service.js*** |  |
| --- | --- |
| **export const** *updateTuit* = **async** (tuit) => {  **const** response = **await *axios***  .put(**`**${TUITS\_API}**/**${tuit.\_id}**`**, tuit);  **return** tuit;  } | *// service function accepts tuit to send server*  *// send HTTP PUT request appending tuit's ID*  *// to URL, and embed tuit object in BODY*  *// return tuit update to update in reducer's*  *// state's store* |

The ***updateTuitThink*** puts the updated ***tuit*** to the server using the corresponding ***updateTuit*** service and the new tuit is dispatched to the reducer to be updated in the state.

| ***tuits-thunks.js*** |  |
| --- | --- |
| **export const *updateTuitThunk*** =  *createAsyncThunk*(  **'tuits/updateTuit'**,  **async** (tuit) =>  **await** service.*updateTuit*(tuit)  ) | *// create update tuit thunk*  *// unique identifier*  *// accepts updated tuit*  *// sends updated tuit to server with service* |

In the previous assignment you implemented the ***TuitStats*** component that rendered a tuit's number of likes and replies. The example below illustrates how the ***likes*** property can be updated from the ***TuitStats*** component. Under the tuit's text, the likes property is rendered along with a heart icon with an ***onClick*** event handler. Clicking the icon dispatches the ***updateTuitThunk*** passing it an updated count for the likes.

| ***tuit-stats.js*** |  |
| --- | --- |
| ...  {tuit.**tuit**}  <**div**>  Likes: {tuit.**likes**}  <**i onClick=**{() => dispatch(***updateTuitThunk***({  ...tuit,  **likes**: tuit.**likes** + 1  })} **className="bi bi-heart-fill me-2 text-danger"**></**i**>  </**div**>  ... |  |

A better implementation might be to let the server update the likes count since there can be thousands of usres liking the same tuit. But since it's just us using the application, this implementation will do for now. The reducer updates the state replacing the old tuit with the updated one and the new state makes its way through the store, provider, selector and back into the user interface for rendering.

| ***tuits-reducer.js*** |  |
| --- | --- |
| [***updateTuitThunk***.**fulfilled**]:  (state, { payload }) => {  state.**loading** = **false**  **const** tuitNdx = state.**tuits**  .findIndex((t) => t.**\_id** === payload.**\_id**)  state.**tuits**[tuitNdx] = {  ...state.**tuits**[tuitNdx],  ...payload  }  } | *// when server update is done*  *// payload contains updated tuit*  *// clear loading flag*  *// find index of updated tuit in array*  *// merge old tuit with updated tuit* |

## 6.5 Exercises

1. Implement ***tuits-service.js*** as described in this section
2. Implement ***tuits-reducer.js*** as described in this section
3. Implement ***tuits-thunks.js*** as described in this section
4. Implement ***tuits-list.js*** as described in this section
5. Using the ***likes*** implementation as a guide, implement a ***dislikes*** counter that renders how many times the tuit has been disliked. Use a thumbs down icon as a button that invokes the dislike behavior. The likes and dislikes counter are independent of each other.

# 7 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 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.

## 7.1 Configuring Node applications to run remotely on Heroku

We're going to host our Node HTTP server on Heroku and we can't use port 4000 when running on Heroku. Instead Heroku declares the proper port to use in an environment variable called ***PORT*** available from Node using ***process.env.PORT***. Refactor ***app.js*** so that it uses the ***PORT*** environment variable if available on Heroku, or uses 4000 otherwise when running locally on our machines.

| ***app.js*** |  |
| --- | --- |
| app.listen(***process***.**env**.**PORT** || 4000); |  |

## 7.2 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 the new file ***.gitignore*** if it does not already exist noting 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 |  |

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 ***tuiter-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/tuiter-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** |
| --- |

## 7.3 Deploying Node server 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 ***tuiter-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., ***tuiter-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 users and the tuits and confirm that both arrays display correctly, e.g., ***https://tuiter-node-server-app.herokuapp.com/api/users*** and ***https://tuiter-node- server-app.herokuapp.com/api/tuits***. ***NOTE:*** the actual URL might be different based on the actual name you chose for the application.

## 7.4 Integrating Netlify React applications with Heroku 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 locally, but it should use the remote Heroku Node server when it's also running remotely on Netlify. To test the remote connection works, refactor ***tuits-service.js*** so that it uses the remote Heroku server instead.

| ***tuits-service.js*** |  |
| --- | --- |
| **import *axios* from "axios"**;  **~~const~~** ~~TUITS\_API =~~ **~~'http://localhost:4000/api/tuits'~~**~~;~~  **const** TUITS\_API = **'https://tuiter-node-server-app.herokuapp.com/api/tuits'**; | |

Reload the ***TuitList*** component and confirm that it now renders the tuits from the server instead of the ones from the local server. 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. Environment variables must start with ***REACT\_APP\_*** and can be accessed from the React code using ***process.env*** as shown below retrieving environment variable ***REACT\_APP\_API\_BASE***. Locally ***REACT\_APP\_API\_BASE*** can be set to ***http://localhost:4000/api***, but when the application runs remotely it can instead be set to ***https://tuiter-node-server-app.herokuapp.com/api***. This way the local React application will connect to the local Node server, but use the Heroku Node server when running remotely in Netlify. ***NOTE:*** use the actual Heroku URL based on the actual application name.

| ***tuits-service.js*** |  |
| --- | --- |
| **import *axios* from "axios"**;  **~~const~~** ~~TUITS\_API =~~ **~~'http://localhost:4000/api/tuits'~~**~~;~~  **~~const~~** ~~TUITS\_API =~~ **~~'https://tuiter-node-server-app.herokuapp.com/api/tuits'~~**~~;~~  **const** API\_BASE = ***process***.**env**.**REACT\_APP\_API\_BASE**;  **const** TUITS\_API = **`**${API\_BASE}**/tuits`**; | |

Setting local environment variables varies depending on the operating system. On ***macOS*** and other ***Unix*** based operating systems you can declare environment variables in ***~/.bash\_profile*** as shown below.

| ***.bash\_profile*** |  |
| --- | --- |
| export REACT\_APP\_API\_BASE=http://localhost:4000/api | |

On Windows you can declare environment variables using the ***Environment Variables*** dialog. Press the ***Window*** key and search for ***Environment Variables***. Once in the ***Environment Variables*** dialog, add an environment variable named ***REACT\_APP\_API\_BASE*** set to ***http://localhost:4000/api***. On any operating system, you might need to restart your machine for changes to take effect.

Add, commit, and push your local React code to the remote GitHub repository. Visit the Netlify application dashboard and click on the ***Site settings*** tab, then ***Build & deploy***, and then ***Environment***. In the ***Environment variables*** screen on the right click the ***Edit variables*** button and enter ***REACT\_APP\_API\_BASE*** in the ***VARIABLE\_NAME*** field and type the Heroku Node URL for the value, e.g., ***https://tuiter-node-server-app. herokuapp.com/api***, and click ***Save***. Redeploy the React application, preview the Web site, and confirm ***TuitList*** renders the same list of tuits as when it runs locally.

## 7.5 Exercises

1. Deploy the Node.js server to Heroku as described in this section
2. Integrate the React application running on Netlify with the Node.js server running on Heroku as described in this section

# 8 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.

# 9 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 ***a8***. When done, add, commit and push both branches to their respective GitHub repositories. Deploy the new branches to Netlify and Heroku and confirm they integrate. Submit links to both your GitHub repositories as well as the Netlify and Heroku URLs where the branches deployed. Here's an example on the steps:

| ***Create a branch called a8*** |
| --- |
| git checkout -b a8  # 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 "a8 REST fa22"  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 ***a8*** branches deployed to on Netlify and Heroku
2. The link to your new branches in GitHub for the React and Node.js projects