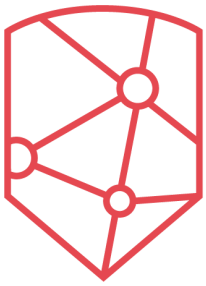
Introduction-2

Introduction



Eleven Fifty Academy

Welcome to the Eleven Fifty Academy Node Server Gitbook

In this book we will be introducing you to the fundamentals of server-side programming and databases, using tools such as Node.js, Express, and PostgreSQL. Some of these concepts may be challenging to understand initially, but practice and experimentation should provide clarity. We'll build a small server in parts, then connect it to a database to store information. Near the end, you'll also build a small client to better understand the relationship of all three parts. You'll also get an idea of how full-stack programming works, as you'll be doing for the rest of the cohort. As always, ask questions and be sure to reflect on what each lesson means in relation to the rest of what you've learned.

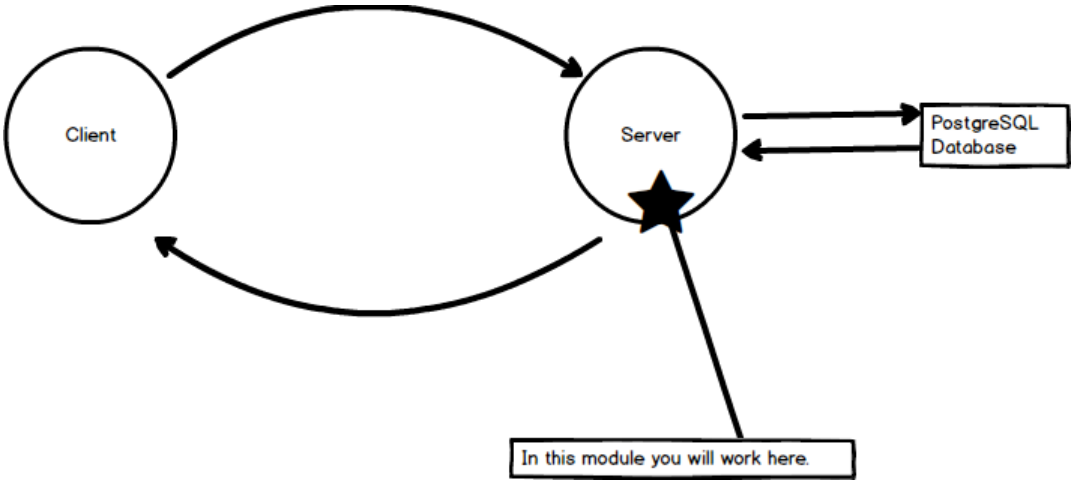
Purpose

01 - Purpose

In this book we'll learn to build a custom server using Node, Express, and PostgreSQL.

Client/Server Model

In the API lessons and challenge, we interacted with the client side. Here, we will be connecting and building on the back-end using Postman and PostgreSQL.



Big Objectives

There will be a lot covered in this chapter. As you proceed, you'll start to see how things are connected, but slow down and take your time if you have trouble comprehending any of it.

Key Objectives:

- Build your own API endpoints and routes with Express
- Use tools to test those endpoints
- Start to use SQL statements and Sequelize to enter and retrieve information into databases
- Set up security protocols and authentication for individual user accounts
- Build a demo api for a Workout App that implements MVC patterns and includes proper routing, authentication, and persistence

Back-End Setup

02 - Back-End Setup

In this module, we'll set up our file structure for our Server.

File Structure

Create the following folders and files in your JavaScript library:

```
Node-Server
    └── server
        └── .gitignore
        └── app.js
```

.gitignore

1. Add `node_modules/` to your `.gitignore` `file`. Again, anytime you work with npm packages, it's best practice to include this in your `.gitignore`. If you forget, hundreds or thousands of extra, unnecessary files will be pushed to your GitHub repository. Simply add the following to the `.gitignore` file: `node_modules/`
2. `app.js` can be left blank for now.

Terms Cheat Sheet

03 - Terms Cheat Sheet

In this module, we'll introduce you to some common terms found in Server/API.

Server Terms Cheat Sheet

The following is a cheat sheet for your usage as you go through this portion of the program. There is not a need to memorize these. Our hope is that you can understand all of them by the time we are done:

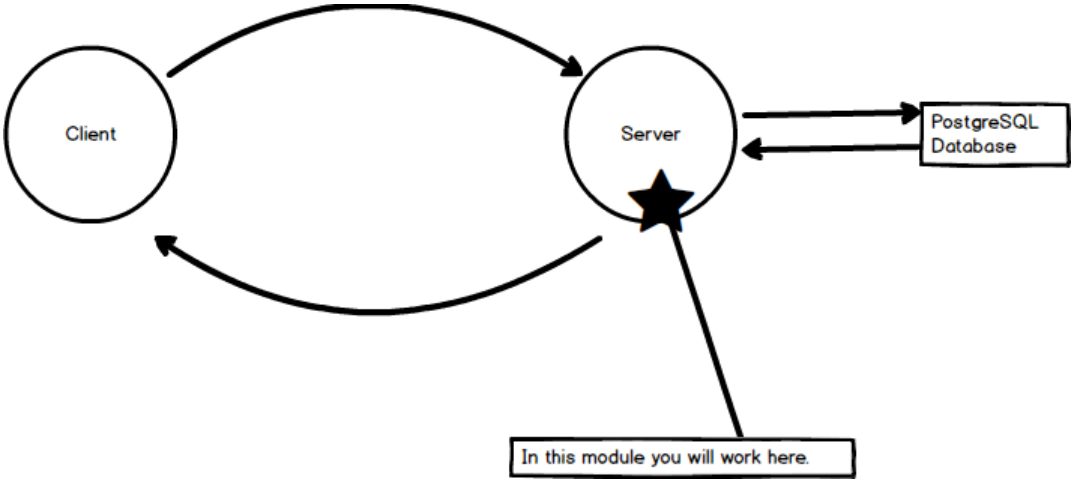
Concept	Definition
server	a program (or hardware) that accepts and responds to requests from a client
client	a program (or hardware) that accesses a service made available by a server
middleware	software that enables communication of data between domains -> For example, www.myapp.com can talk to www.apiformyapp.com
router	determines how to handle an incoming HTTP request
route	the actual path that handles a request: `localhost:3000/helloworld`
controller	receives the request from the router and decides what to do with it
endpoint	a function like POST or DELETE available through shared base routes: `/user/post-data` or `/user/delete-data`
database	a structured set of data held that is accessible in various ways
database table	used in relational databases to store data; a flat file with vertical columns and horizontal rows
data model	server code that mirrors the structure of a database table
encryption	the conversion of data into encoded data to hide its true value: `"mypassword"` -> `"2XKLJlkjasdlfl83"`
statelessness	the idea that a client and server forget each other after each request/response lifecycle
session	a timed meeting between two different devices or applications
token	allows a server to identify a client more easily between requests
authentication	the process of comparing credentials provided by a client with those found in a database
SQL	a language used to query a relational database
migration	the process of updating a database table if a data model changes

npm packages

01 - npm packages

In this module, we'll set up a few of the server packages (dependencies) that will be used in this application.

Orientation



What is a Dependency?

A dependency is a necessary requirement for a class or interface to use.
Think of this analogy:

- If you have a TV remote, you also need the batteries.
- You cannot use the remote without also using those batteries.
- Furthermore, you cannot continuously reuse that remote unless you also continuously reuse those batteries.
- If your batteries fail, you can no longer use your remote.

Just like the remote depends on batteries. Your application will have other applications and frameworks (like Express) in the form of packages to rely on for it to run.

npm init

Let's initialize `npm` as our package manager for our dependencies. To initialize npm in the application, go through the following steps:

1. Open the `server` folder.
2. Open the command line window.
3. Run `npm init`.
4. Hit enter through all the prompts.
5. You should see a new `package.json` file in your `server` folder.

package.json

1. Take out the contents of the `package.json` file.
2. Replace the contents with the code below.
3. Run `npm update`. This will read all of the new dependencies from the `package.json` and load them all in the node_modules folder.
4. Know that adding these new packages to `package.json` file allows us to freeze the particular versions of each dependency so that they harmonize together without breaking the application. For instance, let's say that dependency bcrypt has changed to `3.0.0` and the new version does not work well with sequelize `4.0.0`. Such a change might break our app.

```
{
  "name": "workoutlogserver",
  "version": "1.0.0",
  "description": "server for workout log app",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node ./node_modules/http-server/bin/http-server"
  },
  "author": "YOUR NAME HERE",
  "license": "MIT",
  "dependencies": {
    "bcryptjs": "^2.4.3",
    "body-parser": "^1.18.2",
    "dotenv": "^5.0.1",
    "express": "^4.16.3",
    "jsonwebtoken": "^8.2.0",
    "pg": "^7.4.1",
    "pg-hstore": "^2.3.2",
    "sequelize": "^4.37.0"
  }
}
```

Project Dependencies

Here is another quick cheat sheet for the dependencies that will be used in this application.

Package	Purpose
bcrypt	Password-specific hashing that protects your password from being stored in plain text
body-parser	Parses the body of an HTTP request before it reaches an endpoint's functionality
dotenv	Allows developers to safely store configuration data
express	A web framework for Node.js that allows routing and processing HTTP requests
jsonwebtoken	A compact and self-contained way for securely transmitting information between parties
pg	A dependency for working with Postgres
pg-hstore	A dependency for working with Postgres for advanced functions with Postgres
sequelize	A tool that allows us to map models, pass data, and complete queries with a database

This just a quick overview of what these do. We'll go more in-depth on some of these packages as we build our server.

Express Intro

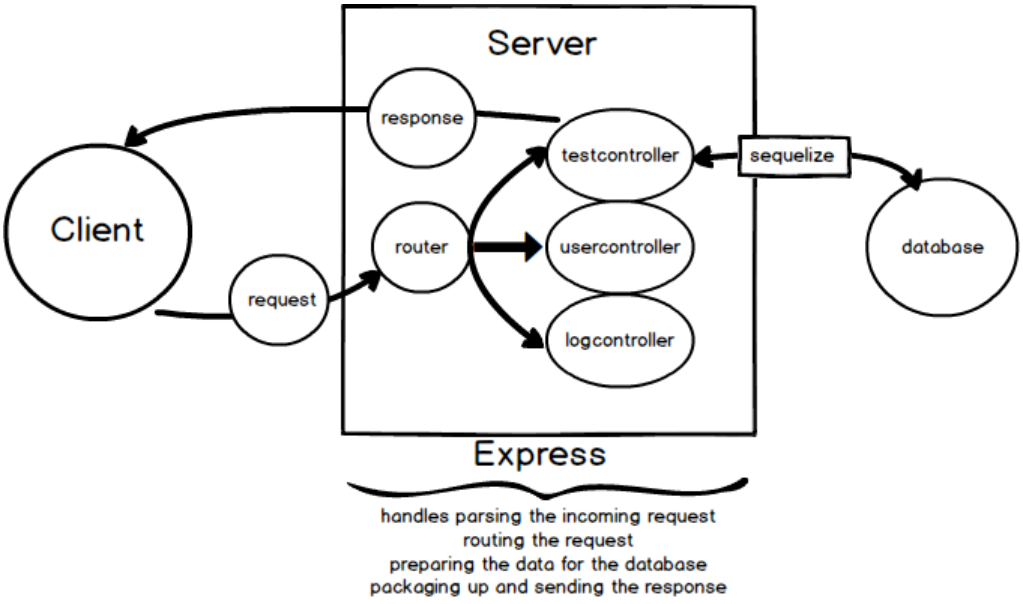
02 - Express Intro

In this module, we'll introduce you to the most important dependency in our server, which is Express.

Express Overview

Express is a Node.js framework that provides developers with tools for building web and mobile applications. Express is well known for making it extremely quick and easy to build APIs. That's what we'll be doing: building an API.

Let's show you a useful overview of some big picture things that Express will help you to handle. We'll refer to this diagram as you move through these lessons:



Express Code

03 - Express code

In this module we'll make a simple Express server.

File Structure

Let's go inside of `app.js` in your `server` directory.

```
Node-Server
    └── server
        └── .gitignore
        └── app.js
```

Basic Server Code

In `app.js`, go ahead and add the following code:

```
var express = require('express'); //1
var app = express(); //2

//3         //4
app.listen(3000, function(){
    console.log('App is listening on 3000.') //5
});
```

Analysis

Let's analyze the code:

1. Here we require the use of the `express` npm package that we've installed in our dependencies.
2. We create an instance of express. We're actually firing off a top-level `express()` function, a function exported by the Express module. This allows us to create an Express app.
3. `app.listen` will use express to start a UNIX socket and listen for connections on the given path. This method is identical to Node's `http.Server.listen()`.
4. The given path is `localhost:3000`.
5. We call a callback function when the connection happens with a simple `console.log`.

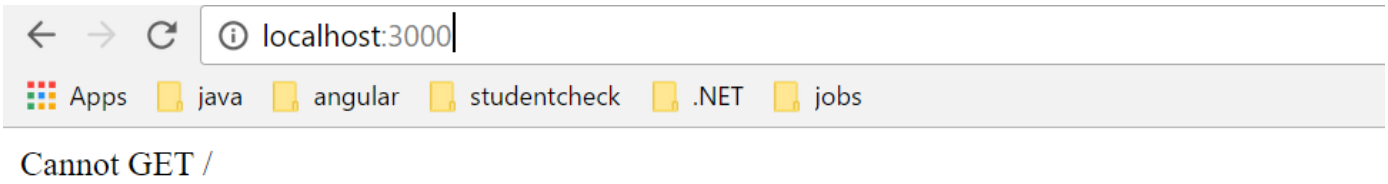
Running the Code

1. Open your terminal in VS Code. Make sure you are in the server directory.
2. Type in `node app.js`.
3. In your console/terminal window, you should see your console message:

```
server aaron * = $ node app.js
sequelize deprecated String based operators are now deprecated. Please use Symbol based operators for better security, read more at http://docs.sequelizejs.com/manual/tutorial/qu
erying.html#operators ..\..\..\..\node_modules\sequelize\lib\sequelize.js:242:13
App is listening on 3000.
```

   Don't worry about the error message right now. It won't affect anything.

4. When we go to **http://localhost:3000/** (http://localhost:3000/), we will see the application running like this:



We'll explain this particular screenshot later.

Nodemon Intro

01 - Nodemon Intro

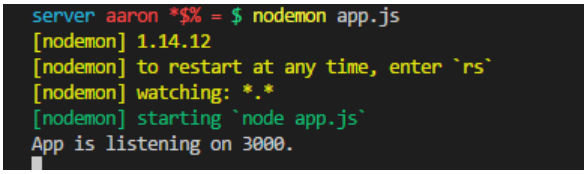
In this module, we'll set up Nodemon for allowing us to more easily make changes to our server code.

Description

When you make a change to code on the client side, you just have to refresh the web page to see the effect of that change. With a server, however, it gets a little more complicated. Any changes that you make won't take effect until you stop and restart the server. When you're testing routes or endpoints and making minute changes over and over, that constant stop/start routine can get kind of ridiculous. Fortunately, we have a tool that will automatically restart your server every time you save a file: Nodemon.

Setup

1. In any directory, run `npm install -g nodemon`. This installs Nodemon globally on your machine.
2. Go into the server directory (where the `package.json` file is visible) run `npm install --save-dev nodemon`. This saves Nodemon to your `devDependencies` in `package.json`.
3. Still in the server directory, run `nodemon app.js`. This should start the server up. You should see this:



4. The app is now running. You can stop the app with `ctrl + c`.
5. Practice running the app with Nodemon by starting it back up with the command. You can push up arrow while in your terminal to get it back quickly.
6. With the server running and with Nodemon started, change the console.log statement in `app.js` to a new phrase. Try using this:

```
app.listen(3000, function(){
 console.log('Hey man!!!')
});
```

7. Press save. `nodemon` should fire back up, and you should see the new `console.log()` statement in your terminal.

Postman Intro

02 - Postman Intro

In this module, we'll introduce Postman, a tool for testing our back-end as we build it.

About Postman

Building a back-end is a little more abstract than building a front-end. In a web application, we can see changes as we make them: a color, a button, an image. With the back-end, we need a way to test our code as we build it to ensure that we are on the right track. Postman is a great tool for that job.

Helpful Features

Postman provides back-end developers with an enormous amount of value, including the following items:

1. Ability to test endpoints.
2. Can fully log in to an application and save data to a database without needing sign in or sign up forms.
3. Code can be written and run directly inside postman to control external servers, devices, etc.
4. Collections of API endpoints can be saved and tested simultaneously.

Install

1. Go to **getpostman.com** **(https://www.getpostman.com/)** .
2. Click `Download the App`.
3. Choose your platform, Windows or Mac.
4. The download should begin. Follow the steps until you have the app opened.
5. Be sure to pin the app in to your toolbar/ribbon. You will be using it extensively in these lessons and in the future.

Postman Setup

03 - Postman set up

In this module, we'll start to use Postman to test our application.

Adding a Response

Let's use Postman to start testing our server by following these steps. Please know that a little bit of the code will be explained in future modules. Right now, we just want to get used to using Postman:

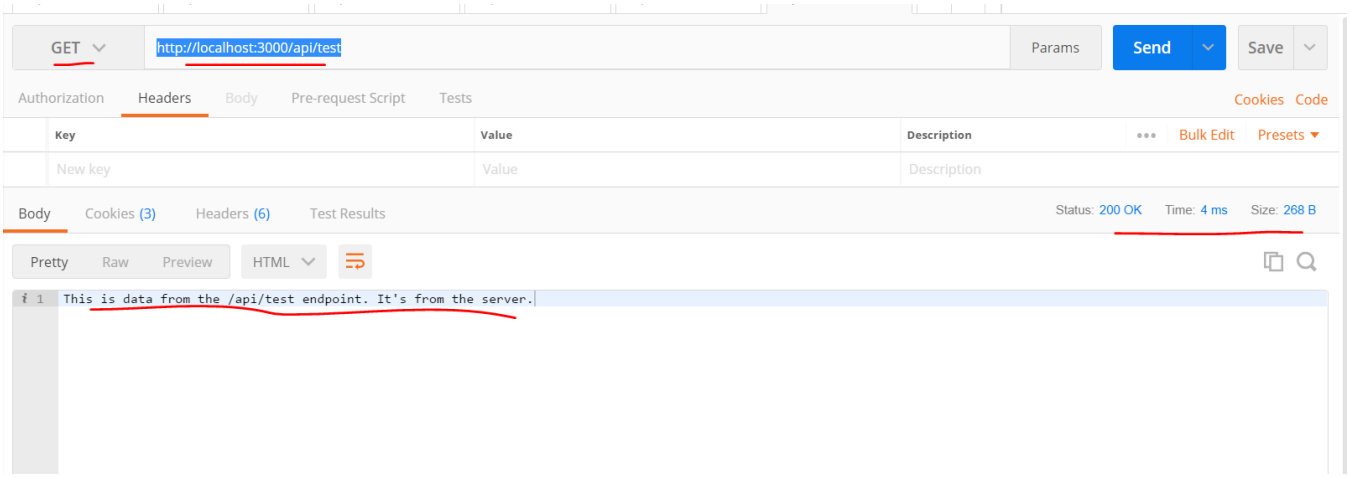
1. Go into the directory with `app.js` and enter the `nodemon app.js` command to start the server. Leave it running.
2. Let's go into the `app.js` file. Add the code below to the bottom of the file:

```
app.use('/api/test', function(req, res){
 res.send("This is data from the /api/test endpoint. It's from the server.");
});
```

3. Again, we'll talk about the code soon. Keep the server running for now.
4. Open up Postman.
5. Choose `File` -> `New` -> `Request`.
6. In the request view, choose `GET` from the dropdown.
7. Enter the following url in the url input field:

   `http://localhost:3000/api/test`

8. Click send.
9. You should see the following result in the Postman response window:



Routes Intro

01 - Routes intro

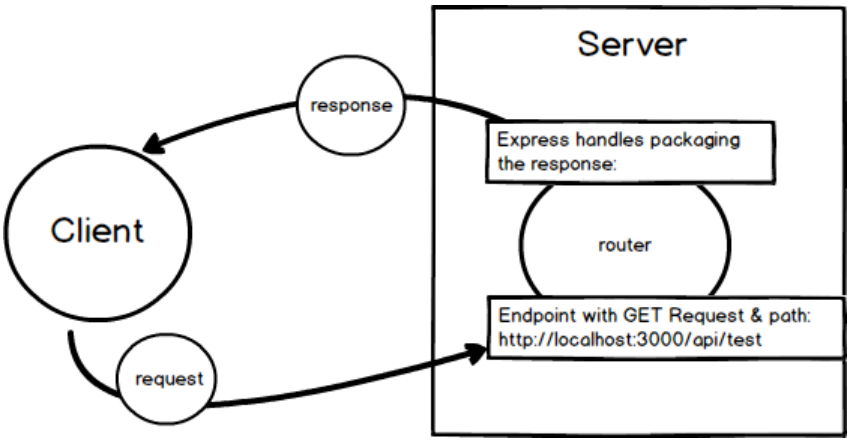
In this module, we'll introduce the concept of routes.

Routing Defined

Routing refers to determining how an application responds to a client request to a particular endpoint. An endpoint is a path and a specific HTTP request method attached to that path (GET, POST, DELETE, PUT).

Overview

Here's a rough diagram of what's happening with our recent code that was used as a test in Postman:



Analysis

1. A GET request is made to localhost:3000/api/test.
2. When the route is requested, Express finds the method for the specific route. The method that we already added for testing Postman is below:

```
app.use('/api/test', function(req, res){
 res.send("This is data from the /api/test endpoint. It's from the server.");
});
```

3. So when we go to the `/api/test/` endpoint, we fire off an Express function `res.send`.
4. `res` (short for `response`) handles packaging up the response object.
5. The `.send()` method does the job of sending off the response.

Express Router() Intro

02 - Express Router() intro

In this module, we'll create a few Express routes. Because an application can have dozens upon dozens of routes, the purpose of this is to help learn to create and organize our routes properly.

File Set up

Let's start by adding a `controllers` folder and a `testcontroller.js` file. We'll explain what controllers are later, but for now, add these two things:

```
javascript-library
    └── Node-Server
        └── server
            └── controllers
                └── testcontroller.js
            └── app.js
```

testcontroller.js

1. Go into the `testcontroller.js` file.
2. Add the following imports along with the `router.get()` function. Numbers are included for upcoming analysis:

```
var express = require('express'); //1
var router = express.Router();  //2

//3    //4   //5          //6
router.get('/', function (req, res) {
    //7
  res.send('Hey!!! This is a test route!');
});

//8
module.exports = router;
```

Analysis

Let's take a look at what this code is doing. It's somewhat of a repeat of what we did in the last module:

1. We import the Express framework and it inside the variable `express`. This instance becomes our gateway to using Express methods.
2. We create a new variable called `router`. Since the `express` variable(line 1) gives us access into the express framework, we can access express properties and methods by calling `express` + `.`. Therefore, when we call `express.Router()`, we are using the `express` variable to access the `Router()` method.
   The `Router()` method will return a `router` object for us. You can read about it more at the Express **docs (https://expressjs.com/en/4x/api.html#router)**.
3. We use the `router` object by using the `router` variable to get access into the `Router()` object methods.
4. `get()` is one of the methods in the object, and we call it here. This method allows us to complete an HTTP GET request. We pass two arguments into the `.get` method.
5. The first argument is the path. In this case, the path is just a `/`. More on this later.
6. The second argument is a callback function. This is also sometimes called a “handler function”. This function gets called when the application receives a request to the specified route and HTTP method. The application “listens” for requests that match the specified route(s) and method(s), and when it detects a match, it calls the specified callback function.

7. Inside our callback function, we call `res.send()`. `send()` is an express method that can be called on the `res` or response object. Our response parameter is just a simple string.
8. We export the module for usage outside of the file.

app.js

Before this will work, we have to add routes into `app.js`.

1. Go to `app.js`.
2. Add the following code:

```
var express = require('express');
var app = express();
var test = require('./controllers/testcontroller')//1

            //2            //3
app.use('/test', test)
```

Analysis

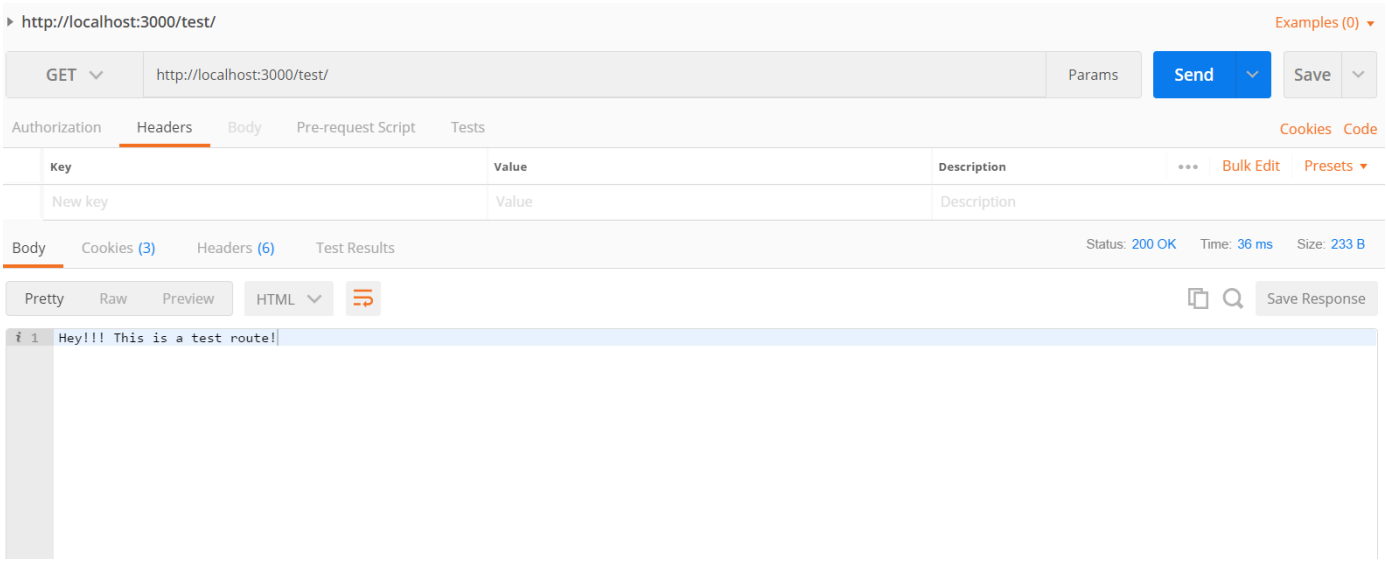
1. We import the route object that we just created and store it in a variable called `test`.
2. We call `app.use` and in the first parameter create a base url called `/test`. So our base url will look like this:

   `http://localhost:3000/test`
3. For our second parameter for the `use()` function, we pass in `test`. This means that all routes created in the `testcontroller.js` file will be sub-routes. It will look like this:

   `http://localhost:3000/test` or `http://localhost:3000/test/`

Test

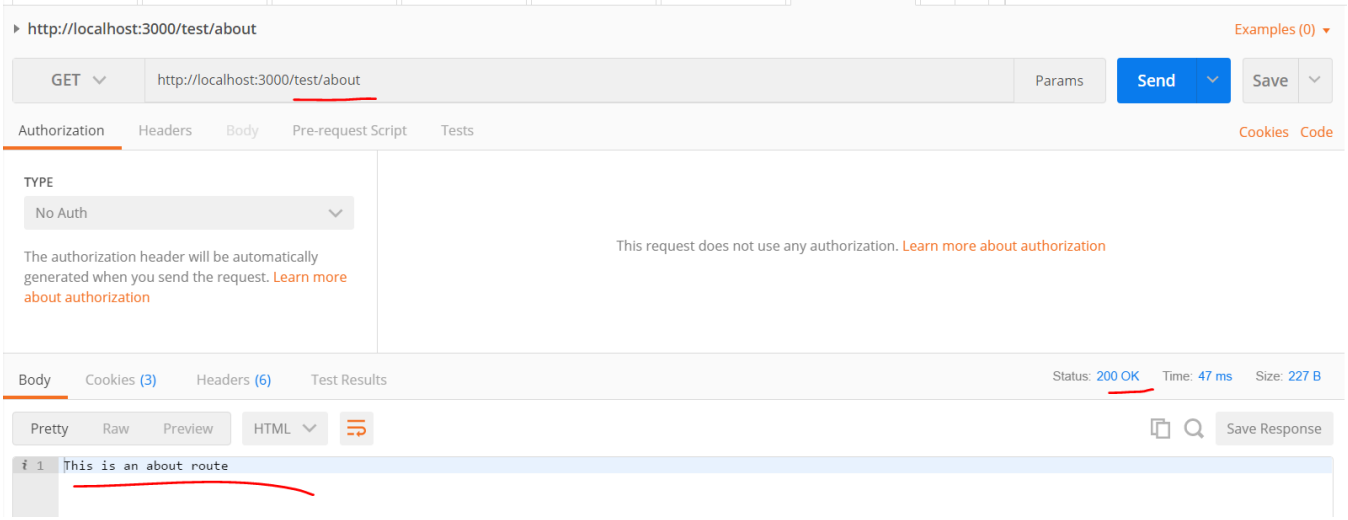
Let's test this now to get a better understanding. 1. Run the application using `nodemon app.js`. 2. Open Postman. 3. In the url link, add the following route into the Request URL bar: `http://localhost:3000/test/` 4. Make sure that you have the request set to a GET request and press SEND. When you send, you should get a response like this:



Challenge #1

We could talk all day about this, but it won't make sense until you play around. Try the following: 1. Create a route that is the following url: `http://localhost:3000/test/about`

1. When you test the app in Postman, you should get a response like this:


http://localhost:3000/test/about
Examples (0)
GET
http://localhost:3000/test/about
Params
Send
Save
Authorization
Headers
Body
Pre-request Script
Tests
Cookies
Code
TYPE
No Auth
The authorization header will be automatically generated when you send the request. Learn more about authorization
This request does not use any authorization. Learn more about authorization
Body
Cookies (3)
Headers (6)
Test Results
Status: 200 OK
Time: 47 ms
Size: 227 B
Pretty
Raw
Preview
HTML
Save Response
1 This is an about route

Challenge 1

03 - Challenge 1

Challenge #1 Solution:

In this module, we'll discuss the solution to the previous challenge and add a few more routes for practice.

About Route Solution

To define the route, you would have done the following in the `testcontroller.js` file:

```
var express = require('express')
var router = express.Router()

router.get('/', function (req, res) {
  res.send('Hey!!! This is a test route!')
});

    //1        //2        //3
router.get('/about', function (req, res) {
  res.send('This is an about route') //4
});

module.exports = router;
```

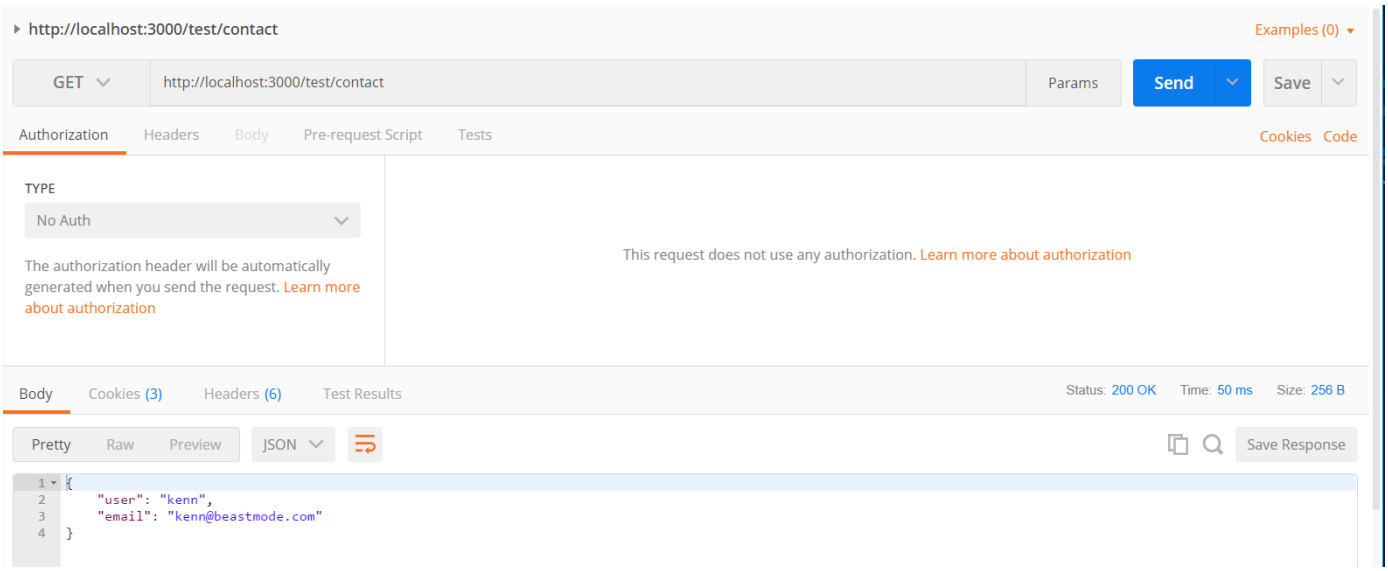
Analysis

Just a few review notes on the challenge: 1. You can use the `router` instance that we've created and call the `.get` method from express to make a HTTP GET request. 2. The first parameter is the `/about` path that we'll be appending to the URL. This will make the url look like this: `http://localhost:3000/test/about` 3. Again, we pass in a callback function that will run when the path is requested. So when we type in the above url, this function fires off. 4. The `send()` method gets called on the `res` object, and a simple string is returned.

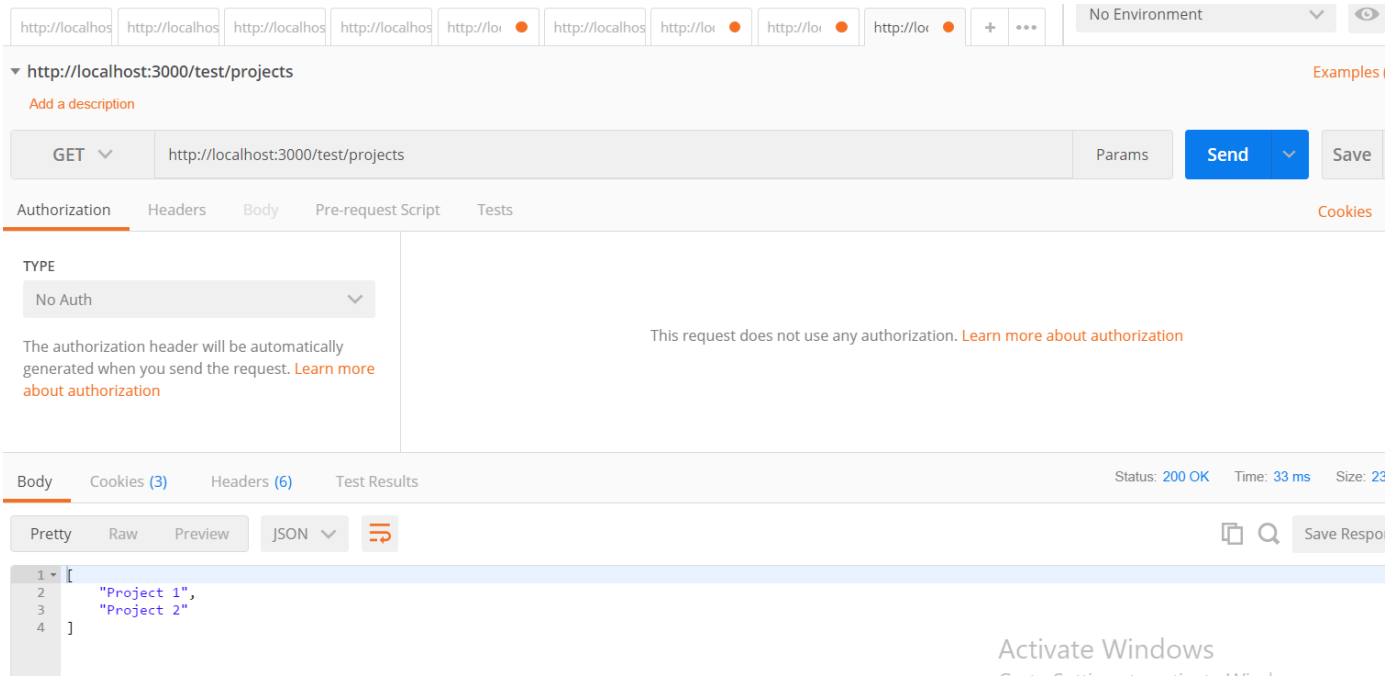
Challenge #2: More Practice Routes

For more practice, add three more test routes with messages of your choice: 1. Create a route that will return a contact object. Here is the path: `http://localhost:3000/test/contact` 2. Create a route that sends an array of projects. Here is the path: `http://localhost:3000/test/projects` 3. Create a route that sends an array of contact objects. Here is the path: `http://localhost:3000/test/mycontacts`

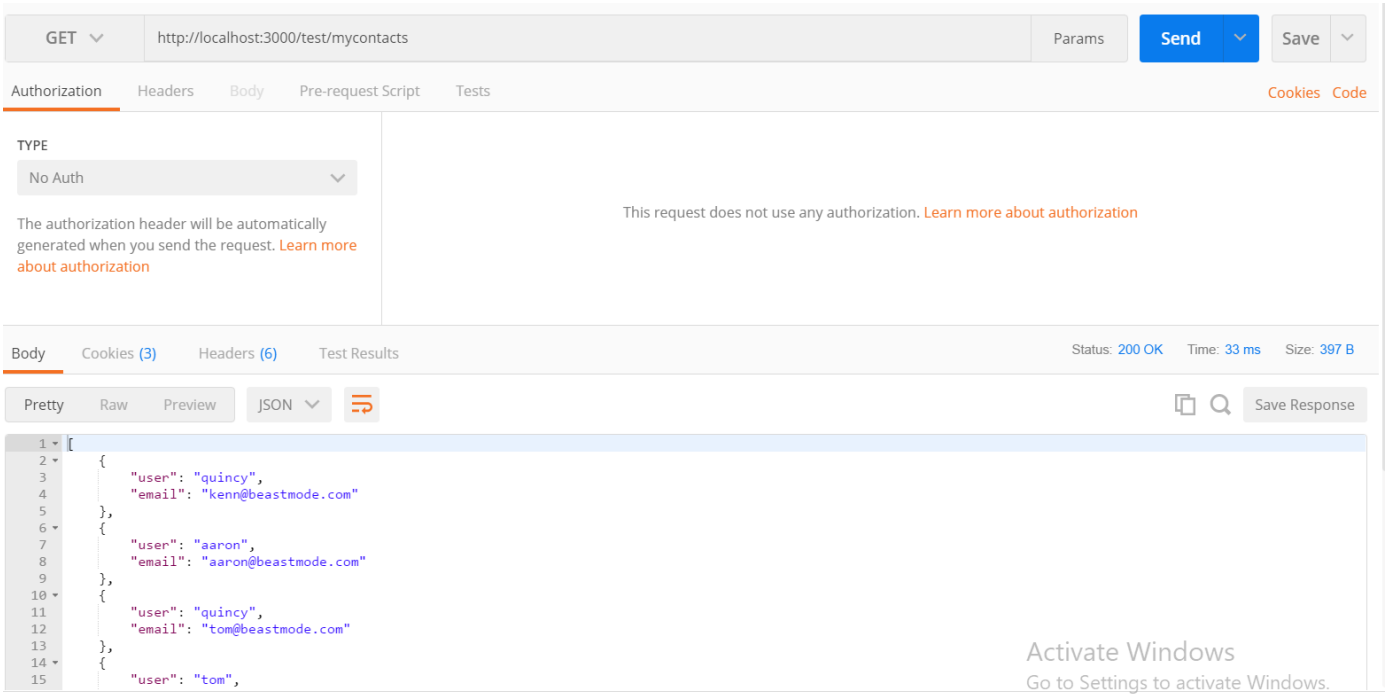
Here is the screenshot for #1:



Here is the screenshot for #2:



Here is the screenshot for #3:


GET
http://localhost:3000/test/mycontacts
Params
Send
Save
Authorization
Headers
Body
Pre-request Script
Tests
Cookies
Code
TYPE
No Auth
The authorization header will be automatically generated when you send the request. Learn more about authorization
This request does not use any authorization. Learn more about authorization
Body
Cookies (3)
Headers (6)
Test Results
Status: 200 OK
Time: 33 ms
Size: 397 B
Pretty
Raw
Preview
JSON
Save Response
[
{
"user": "quincy",
"email": "kenn@beastmode.com"
},
{
"user": "aaron",
"email": "aaron@beastmode.com"
},
{
"user": "quincy",
"email": "tom@beastmode.com"
},
{
"user": "tom",
Activate Windows
Go to Settings to activate Windows.

Challenge 2-2

04 - Challenge 2

In this module, we give you the answer to the challenges. Spoilers ahead.

Here are the routes to add to the `testcontroller.js` file:

```
var express = require('express')
var router = express.Router()

router.get('/', function (req, res) {
  res.send('Hey!!! This is a test route!')
});

router.get('/about', function (req, res) {
  res.send('This is an about route');
});

//1 Pass in an object
router.get('/contact', function (req, res) {
  res.send({user: "kenn", email: "kenn@beastmode.com"});
});

//2 Pass in an array
router.get('/projects', function (req, res) {
  res.send(['Project 1', 'Project 2']);
});

//3 Pass in an array of objects
router.get('/mycontacts', function (req, res) {
  res.send([
    {user: "quincy", email: "kenn@beastmode.com"},
    {user: "aaron", email: "aaron@beastmode.com"},
    {user: "quincy", email: "quincy@beastmode.com"},
    {user: "tom", email: "tom@beastmode.com"}
  ]);
});

module.exports = router;
```

DB Intro

00 - DB Intro

DATABASES

In order to get deeper into routes and the server, we'll need to start working with a database. In this module, we'll discuss two types of databases, examples, and some common terms associated with them.

Database types

There are two types of databases that we'll discuss here:

1. Relational Database:

A relational database is structured to recognize relations among stored items of information.

Rows and Columns

A relational database stores data in a table-like display using rows and columns. Think about it like an Excel spreadsheet:

ID	Players
1	Durant
2	Barkley
3	Miller

Primary Key

Every table has something called a "primary key". Usually this "primary key" is the ID column. Consider a *Breakfast of Champions* table:

ID	Product
1	Kale
2	Broccoli
3	Cigarettes

Each row will have a unique ID that is auto-generated incrementally when it is added to Postgres. More on that later.

SQL

In addition, relational databases use SQL (Structured Query Language) to manage the information. SQL is most often pronounced "Sequel", but you may also hear pronounced "S - Q - ELL". SQL is a language of its own, using commands to create, delete, change, or display information from the database. Relational databases are often called "SQL databases" for this reason. We will teach you about SQL in a future lesson.

2. Non-Relational Databases

A "non-relational database", or "No-SQL" database, is usually structured like a JSON object. Rather than having tables, everything is set up like an object. Different items can have a different number of properties or even different properties altogether. This can be useful when dealing with a large or diverse set of data. Additionally, no SQL statements are used, which helps prevent many common types of attacks against databases. Some examples of these are Firebase from Google and MongoDB.

Example: Let's look at our person object in this case. We would set up the data like this:

```
{
    "person": {
        "1": {
            "firstName": "Aaron",
            "lastName": "Ofengender",
            "height": "70in",
```

```
            "eyeColor": "brown",
            "glasses": true
        },
        "2": {
            "firstName": "James",
            "lastName": "Smith",
            "height": "65in",
            "eyeColor": "blue",
            "glasses": false
        }
    }
}
```

So Which are We Using?

We will be using PostgreSQL, a traditional relational database.

PostgresQL Intro

01 - PostgreSQL Intro

In this module, we'll give you a high level perspective of PostgreSQL and PG Admin with a brief history behind it.

PostgreSQL History

PostgreSQL, also commonly called "Postgres", was originally created in 1986 at the University of California-Berkeley by a professor and some of his graduate students. Postgres was one of the first major attempts at creating a relational database system. Over the course of the next decade, it underwent several major updates and revisions within the university, and a version of it was eventually purchased by IBM in 2001. In 1996, however, some developers outside of the university jumped on the open-source project and began a complete re-write of the system, including replacing its native, proprietary language with SQL. This culminated in 2004 with PostgreSQL 6.0, the first version as we know it today. Still one of the most-widely used database systems in the world, it has continually been open-sourced and updated, now currently at version 10.3.

PG Admin

PG Admin was originally created as pgmanager in 1998, a Windows-based GUI tool to manage PostgreSQL databases. Several versions of the program later, PG Admin 4 was released in 2016. Written in C++, it focuses on cloud-based databases while also providing support for local and physical databases.

pgAdmin, now pgAdmin 4, is the most popular Open Source administration and development platform for PostgreSQL.

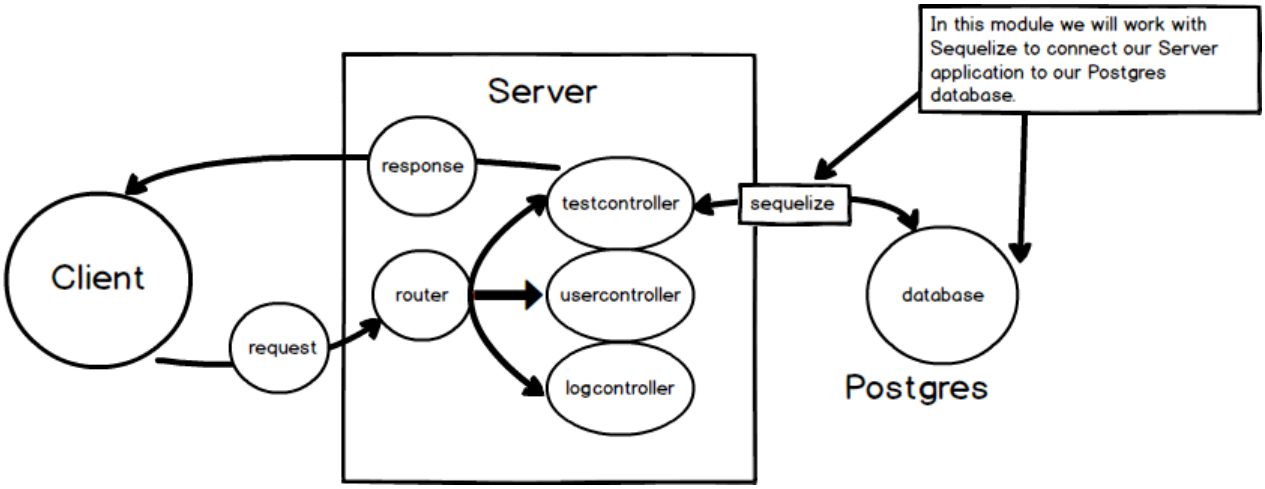
Install

02 - Install

In this module, we'll set up the Postgres database for our server using PG Admin and Sequelize.

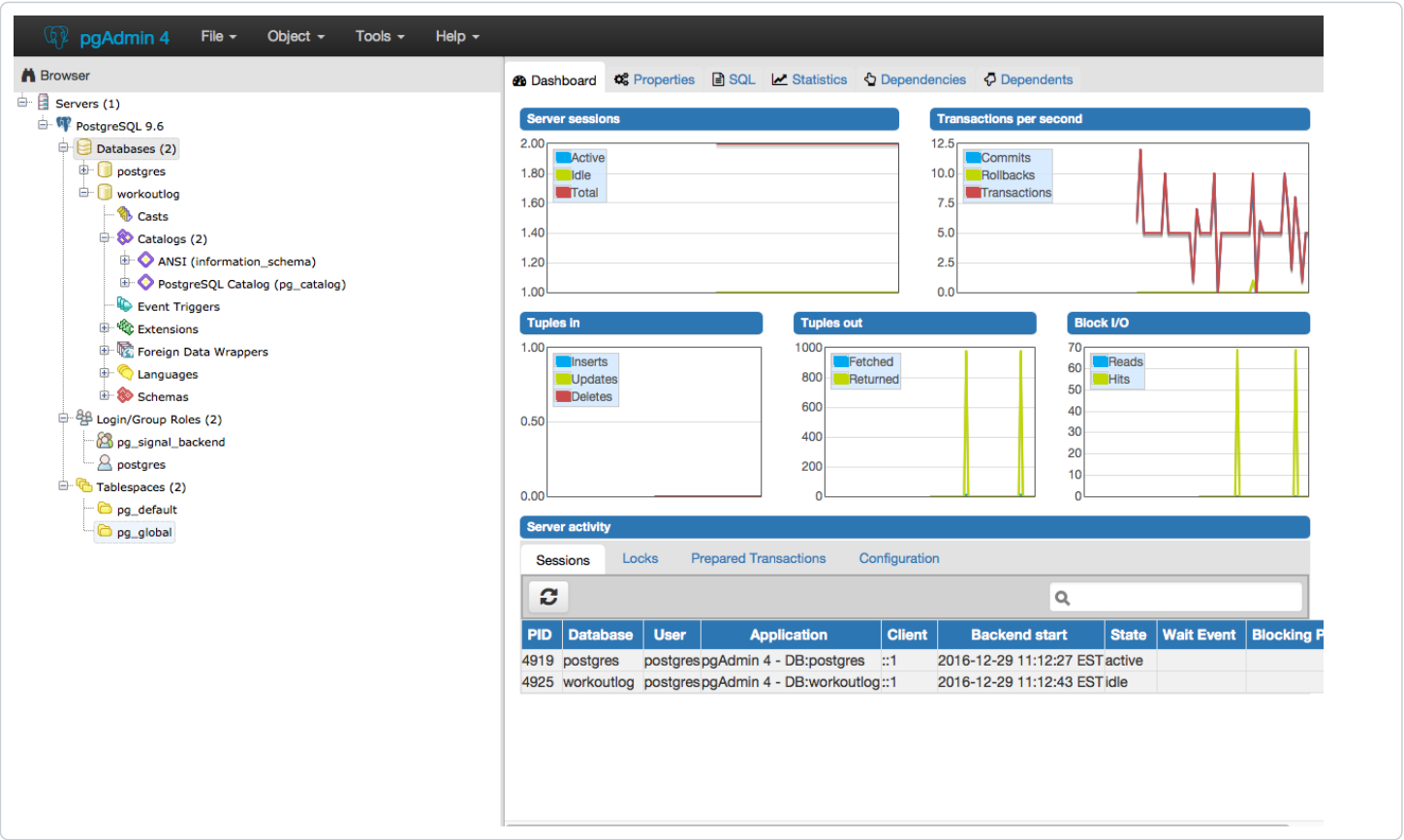
Location

We're going to be working with the database, as shown here:



Install Directions

1. Come up with a Postgres password right now. Make it unique. Don't make it into a password that you use for anything else.
2. **Please take the time to record this password somewhere safe**. Resetting a Postgres password is not impossible, but is very challenging and, more importantly, very time consuming. Take the time to document this password. Several previous students have spent many hours attempting to reset their password. I use `Letmein1234!`. This will be for local use only, so there is no fear of someone else accessing your database using this password.
3. RECORD YOUR PASSWORD SOMEWHERE!!!
4. Install postgres **here** **(https://www.postgresql.org/download/)**.
5. When the download is done, go through the install steps. **NOTE: You should set the port to 5432 (this is the default port number. PLEASE DO NOT CHANGE IT!!).**
6. After the download completes, choose PostgreSQL from the dropdown in Stack Builder.
7. Open up PG Admin 4.
   - Mac users should click on plug
8. Click Server, then click Postgres.
9. Log in with your memorable password.
10. Right click on *"Database"*.
11. Choose *"Create Database"*.
12. Name it "workoutlog".



You've just created your first database! We're finished with the setup, but if you have any problems with this, ask us and we'll help. Before moving on, please make sure that others around you have been able to create a database.

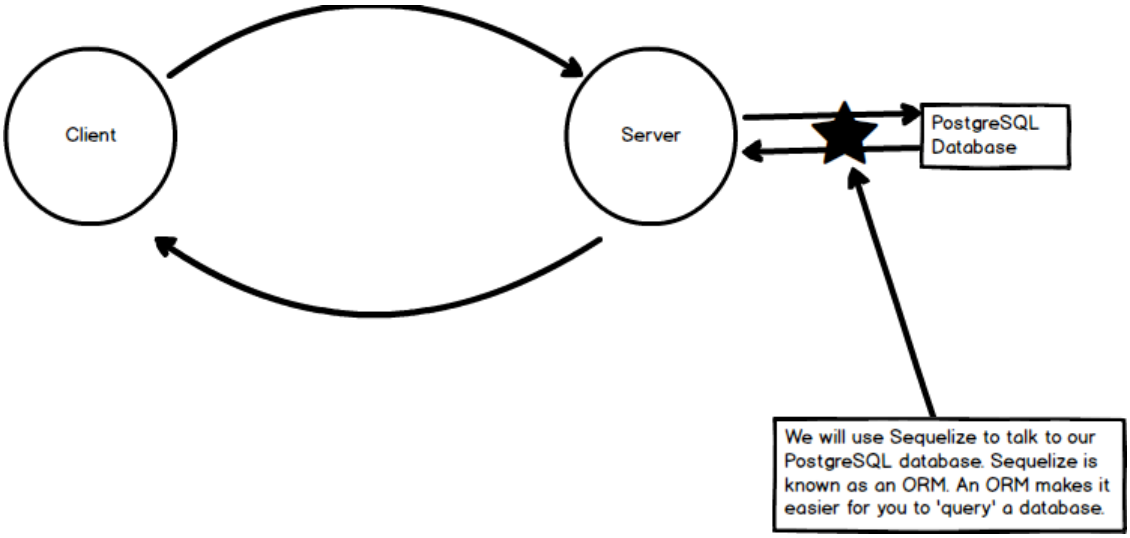
Sequelize Intro

01 - Sequelize intro

In this module, we'll introduce `Sequelize`, a tool that helps us build models to map to our database.

Orientation

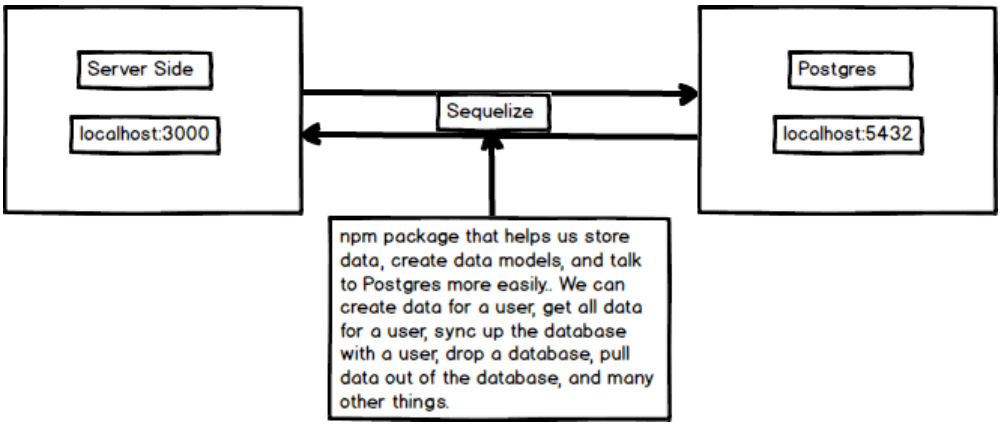
Here is where we'll be:



What is Sequelize?

Sequelize is a promise-based ORM for Node.js. It supports several different SQL dialects, including PostgreSQL, MySQL, SQLite, and MSSQL. Essentially, one of the main features of Sequelize is that it allows developers to communicate between the server and the database more fluidly.

Take a minute to study the following diagram:



As you can see, Sequelize does the work of communicating between the Server application and the database.

Packages

If you look in your `package.json`, you'll see that we have a few packages for using Postgres and Sequelize. The `sequelize` package relies on the `pg` package to connect to Postgres.

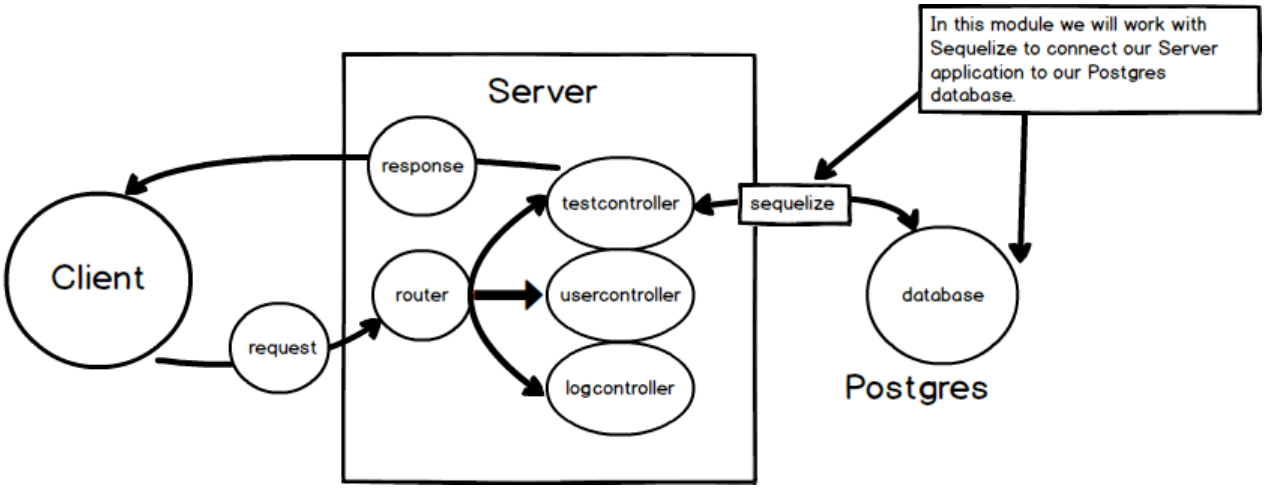
Initialize

02 - Initialize

In this module, we will use Sequelize to connect our server to PG Admin.

Orientation

Right now we'll be working with Sequelize and Postgres:



Sequelize Connection

To use Sequelize, we'll have to establish a connection. This is the standard approach from the **docs (http://docs.sequelizejs.com/manual/installation/getting-started.html#test-the-connection)** , and it is often copy and pasted into projects for setup. Simply put, it's allowing us to connect from our project to the Postgres database. If you haven't already done so, create the file `db.js` inside your `server` folder and put this code inside of it:

```
//1
const Sequelize = require('sequelize');

      //2               //3       //4         //5            //6
const sequelize = new Sequelize('workoutlog', 'postgres', 'Letmein1234!', {
    host: 'localhost', //7
    dialect: 'postgres'  ///8
});
    //9      //10         //11
sequelize.authenticate().then(
    function() { //12
        console.log('Connected to workoutlog postgres database');
    },
    function(err){ //13
        console.log(err);
    }
);
                //14
module.exports = sequelize;
```

Analysis

Let's do some analysis of the code above. You do not need to memorize all of this information. Read through it, and then use it as a reference for when you need it next:

Concept Analysis

Concept Analysis

1 Import the Sequelize package.
2 Create an instance of Sequelize for use in the module with the `sequelize` variable.
3 Use the constructor to create a new Sequelize object.
4 Identify the db table to connect to.
5 The username for the db.
6 The password for the local db.
7 The host points to the local port for Sequelize. This is 5432.
8 Identify the QL dialect being used. Could be MSSQL, SQLLite, or others
9 Use the `sequelize` variable to access methods.
10 Call the `authenticate()` method.
11 `authenticate()` returns a promise. Use `.then()`.
12 Fire a function that shows if we're connected.
13 Fire an error if there are any errors.
14 Export the module.

app.js

We also need to do some configuration in our `app.js` file.

Do the following: 1. Create a `sequelize` variable that imports the db file. 2. Use the variable and call `.sync()`. This method will ensure that we sync all defined models to the DB.

```
var express = require('express');
var app = express();
var test= require('./controllers/testcontroller')
//1
var sequelize = require('./db');

//2
sequelize.sync(); // tip: pass in {force: true} for resetting tables

app.use('/test', test)
```

NOTE: When you run your database with Sequelize, you may see the following message:

```
sequelize deprecated String based operators are now deprecated. Please use Symbol based operators for better security, read more at http://docs.sequelizejs.com/manual/tutorial/querying.html#operators ..\..\..\..\node_modules\sequelize\lib\sequelize.js:242:13
```

Newer versions of sequelize make use of the new `Symbol` datatype instead of relying on strings. You can still use strings for now; this message just means that in the future they may not. We haven't covered Symbols much, but feel free to do some research on your own and try them out.

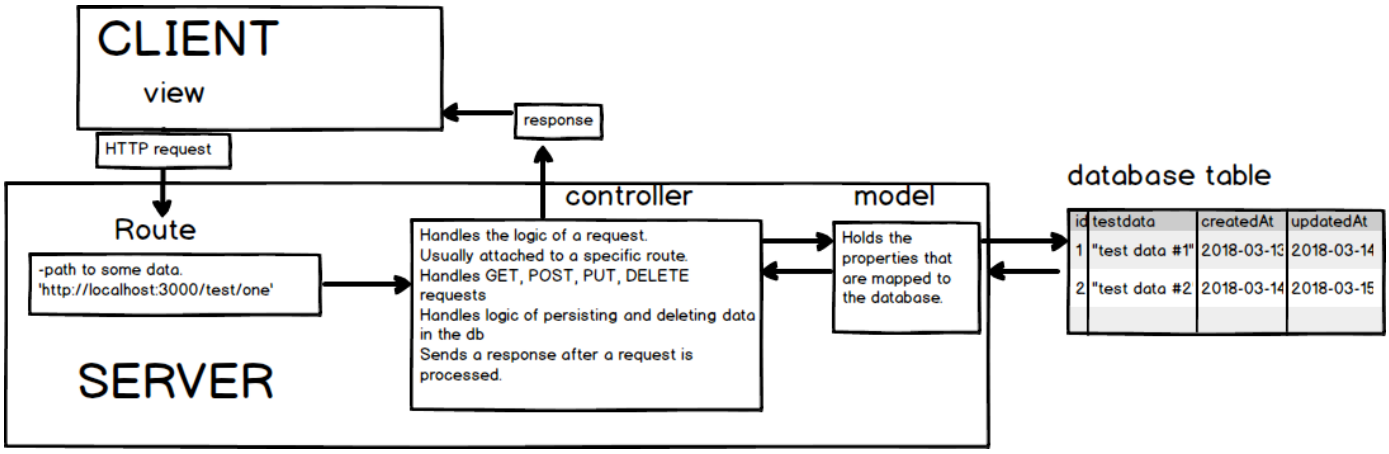
MVC Intro

00 - MVC Intro

In this chapter, we'll walk through several modules that will teach how to build controller methods and routes that make POST requests containing data. Those requests will post data that will persist in our Postgres database and return that data to the client in a response. To write this code, we'll use a Model-View-Controller pattern, so we'll introduce you to MVC at this time.

The Model-View-Controller Software Pattern

MVC is a common software pattern found in many programming languages and applications. Take a look at a diagram of MVC to start getting an idea of how it works:



Views

For this chapter, we won't be dealing with the view, but we'll say that the view/client would be sending requests, receiving responses, and working to display the data to users. For a temporary client, we'll use Postman in lieu of a View.

Controllers

It's good to think of the controller as something that handles the heavier logic in the application. A controller is usually a method or methods that will handle some or all of the following things:

1. Receiving the incoming request depending on the route.
2. Processing the type of incoming request: GET, POST, PUT, DELETE.
3. Collecting the data from the incoming request.
4. Working with the model to ensure that the request data matches the types in the model and the database.
5. Creating, updating, reading, or deleting objects in the database.
6. Sending off the response for the incoming request.

Models

Coding models are usually considered to be just this: `representations of the data being handled`. Models can do the following:

1. Represent the data being stored in the database.
2. Dictate the types of data that will be stored (string, boolean, integer).
3. Handle some basic business logic in an application, such as a character limit for a string being stored in the db, formatting for date and time details, and many other things that shape the data in the database.
4. Used by the controller to handle logic.

Endpoints & Routes

As mentioned in a previous module, when an HTTP request comes in, it hits a route and finds the proper endpoint. When the router finds the proper endpoint, the proper controller method is fired, and the controller method handles any necessary logic. To practice learning about endpoints, controllers, and models, we'll be building the following endpoints in the modules ahead:

```
http://localhost:3000/test/one - POST
http://localhost:3000/test/two - POST
http://localhost:3000/test/three - POST
http://localhost:3000/test/four - POST
http://localhost:3000/test/five - POST
http://localhost:3000/test/six - POST
http://localhost:3000/test/seven - POST
```

When we start up our server, each of these will be available for processing a request. We'll work with each one as a way to learn the process of writing a controller method.

Before we fire off controller methods, we'll start by building some data models.

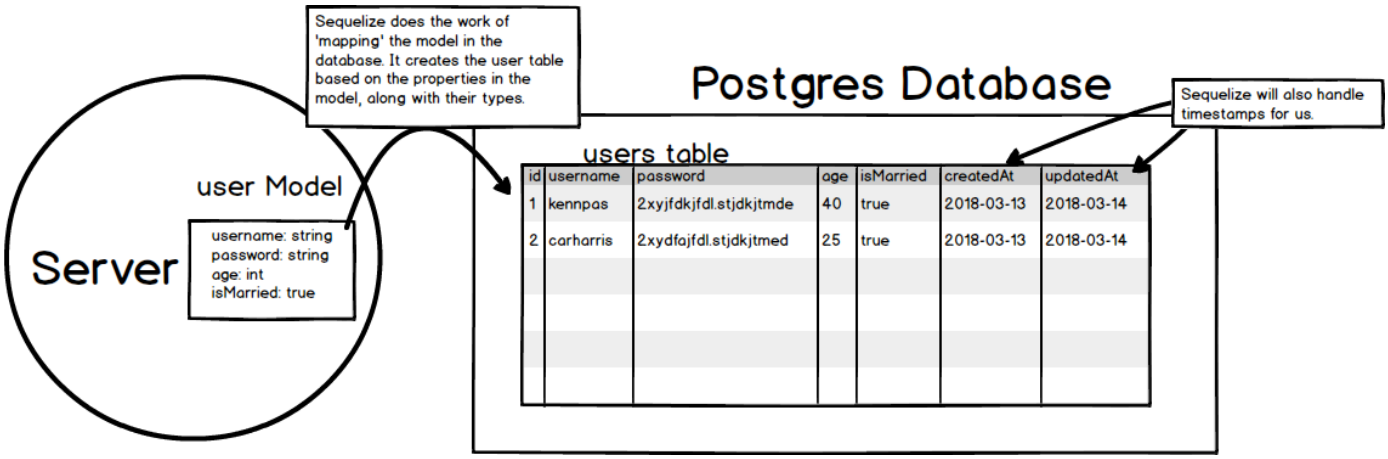
Intro to Models

01 - Intro to Models

In this module, we discuss why we need models in our MVC application.

Rationale

Essentially, we will use models to define the data that we want to store. The model in our server will closely mirror the data table in our database. Consider the following diagram:



Notice how the user model in the server has the following properties: `username`, `password`, `age`, and `isMarried`. These are all properties of a user model that will eventually show up in the database. Notice how the Postgres database table has columns for these properties.

The model and the db table are not exactly the same though. Sequelize will do the following under the hood:

1. Add an id that increments automatically and without repeating (1, 2, 3, 4, etc.)
2. Add a timestamp for when the row was created.
3. Add a timestamp for when the row is updated.

Let's start learning how to configure a model with a db table by building our own test model.

Test Model

02 - Test Model

In this module, we'll introduce you to a Sequelize model and use the `.define()` method to build a database model.

File Setup

Let's start by adding a `models` folder and a `test.js` file:

```
javascript-library
    └── 5-Express Server
        └── Server
            └── controllers
            └── models
                └── test.js
            └── app.js
            └── db.js
```

Test Model

Let's create our first model, a test model:

```
     //7                        //1
module.exports = function (sequelize, DataTypes) {
                       //2   //3
    return sequelize.define('test', { //4
        //5         //6
        testdata: DataTypes.STRING
    });
};
```

Analysis

1. We run an anonymous function that has two parameters: `sequelize` and `DataTypes`. The function will return the value of what is created by `sequelize.define`.
2. We use the `sequelize` object to call the define method. `.define()` is a Sequelize method that will map model properties in the server file to a table in Postgres. You can read more about it **here** **(http://docs.sequelizejs.com/manual/tutorial/models-definition.html)** .
3. In the first argument of the `define` method, we pass in the string `test`. This will become a table called `tests` in Postgres (the table names are pluralized).
4. Our second argument of the define function is an object. Any key/value pairs in the following object will become columns of the table. The syntax looks a little weird here. Just know that it's an object that we can pass in numerous properties to create numerous table columns.
5. `testdata` is a key in our model object that will be a column in our database.
6. `DataTypes.STRING` is our value for the `testdata` property. Because we define it as a `STRING` value in the model, any information to be held in that column MUST be a `string` data-type. Remember that `DataTypes` is a parameter in the function brought in through Sequelize. You can read more about Sequelize DataTypes **here** **(http://docs.sequelizejs.com/manual/tutorial/models-definition.html#data-types)** . Although JavaScript is a loosely typed language, Postgres wants to know what data types we're adding to each of our columns. Sequelize is making us declare the data types that we'll be storing.
7. The model is exported to allow Sequelize to create the `tests` table with the `testdata` column the next time the server connects to the database and a user makes a POST request that uses the model.

Controllers Intro

00 - Controllers Intro

In this module, we'll reiterate the purpose of our controller methods and give an overview of the next seven modules.

Key Points

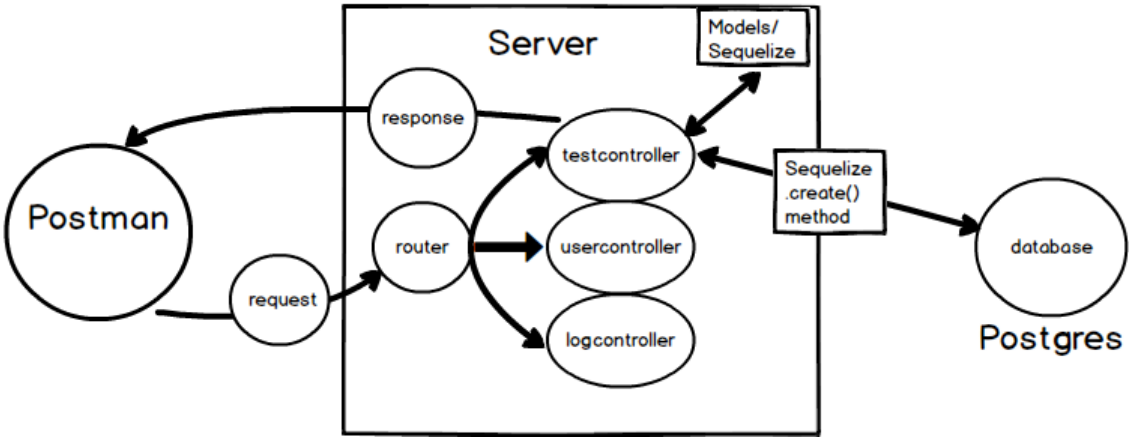
It's good to think of a controller as something that handles or 'controls' the heavier logic in the application. A controller is a method or methods that will handle some or all of the following things:

1. Receiving the incoming request depending on the route.
2. Processing the type of incoming request: GET, POST, PUT, DELETE.
3. Collecting the data from the incoming request.
4. Working with the model to ensure that the request data matches the types in the model and the database.
5. Creating, updating, reading, or deleting objects in the database.
6. Sending off the response for the incoming request.

Final Flow

So far, in previous modules, we've made GET requests with HTTP. We have made requests that ask the server to send us some hard coded data in a response. Now, we want to work towards using Postgres to store data that a user might send to it in a POST request. Clearly, we want to be able to do this so that our users can save data and come back to it later.

Let's look at the final flow of how this will work. We will build towards the items in this diagram as we go:



That will be the completed flow after we create all of our endpoints and controller logic.

Let's get started on discussing how to build controller methods and use them to POST data and persist it to our database.

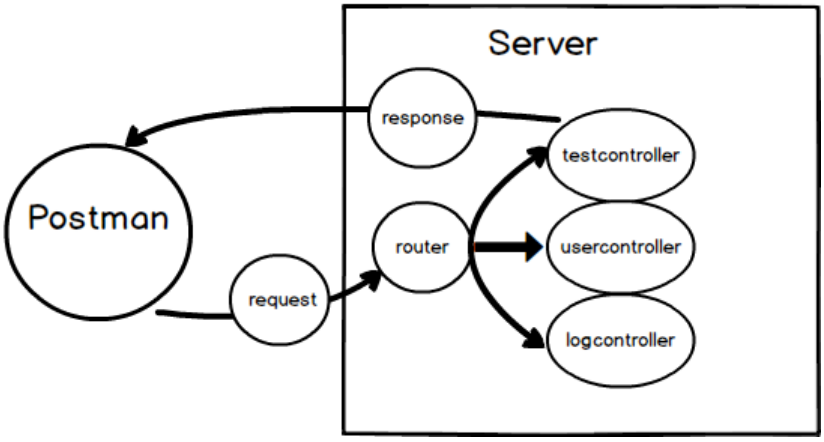
Controller Setup

01 - Controller Set up

In this module, we'll get started on an endpoint that handles a simple POST request.

Overview

Here's the flow of what we will have after this module:



Notice that there is no database or models at this point.

Location

We're going to be adding to our `testcontroller.js` file that we built back in our lessons on routing.

```
Node-Server
    └── controllers
      └── testcontroller.js
    └── models
    └── app.js
    └── db.js
```

Code

Before proceeding, please clear out all previous test methods in `testcontroller.js`. Add the following items to the file:

```
var express = require('express')
var router = express.Router()
var sequelize = require('../db');

/*************************************
 * Controller Method #1: Simple Response
*************************************/
      //1       //2
router.post('/one', function(req, res){
  //3
  res.send("Test 1 went through!")
});

module.exports = router;
```

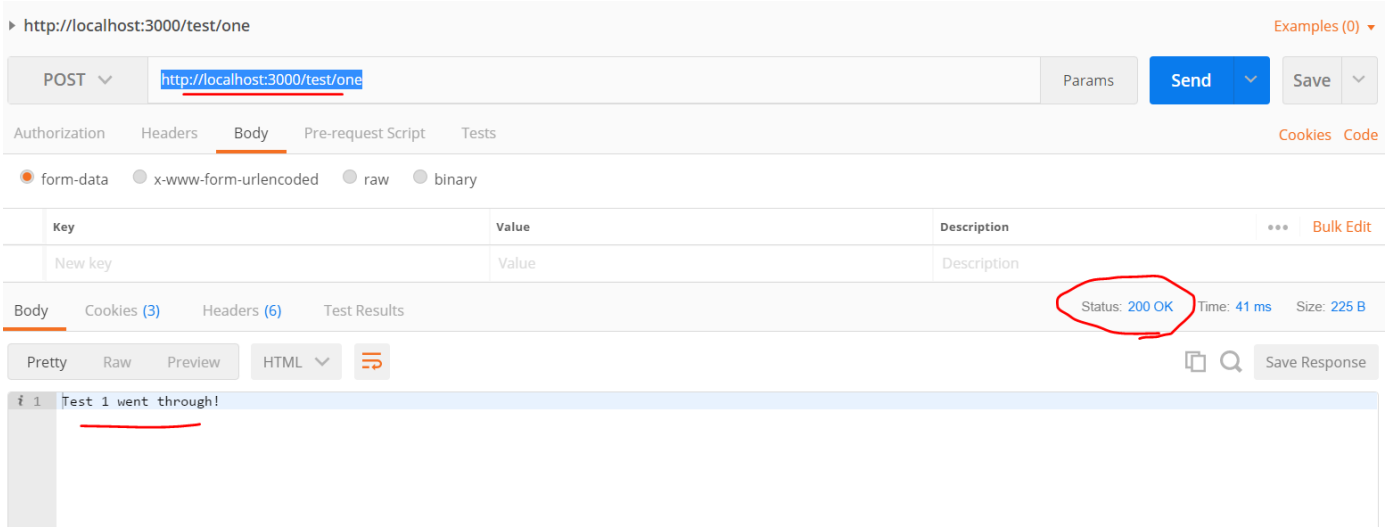
Analysis

1. We use the Express `router` object to call the `post()` method. This corresponds to the type of HTTP request that we are sending. `POST` is telling the server that the incoming request has data coming with it. You use a POST request when you sign up for an application, send an email, send a tweet, post on a wall, etc. POST sends data through HTTP to the server, which might send the data to the database to be stored.
2. `/one` will be the endpoint/route we are using. Our route will be named `http://localhost:3000/test/one`. After that, we'll run a callback function, which will fire off a response.
3. When the client requests the given endpoint, we simply send a string in our response.

KEY POINT: Notice that we are not yet talking to our model or database. We are simply sending an empty POST and returning a string response.

Test

Let's test this in Postman. 1. Make sure your server is running. 2. Open Postman. 3. Open a new request. 4. Change the dropdown to POST. 5. Enter the endpoint into the URL input field: `http://localhost:3000/test/one` 6. Press 'Send'. 7. You should see the following response:



Summary of the Flow

In this module, the following flow is happening:

1. We make a POST request with Postman.
2. The router sends that request to the `testcontroller`.
3. The `testcontroller` method fires off a callback with a response.
4. The callback sends back the response to Postman.

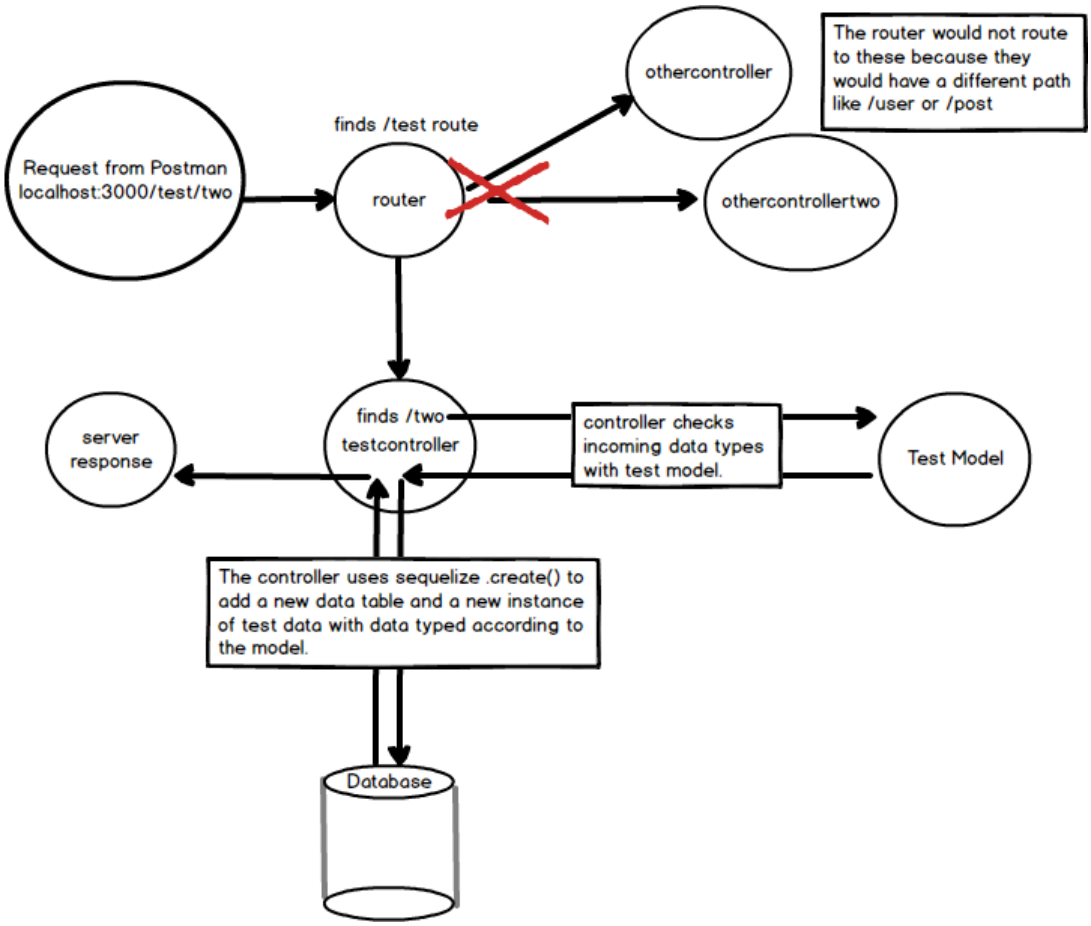
Create Method

02 - Create Method

In this module, we're going to create a second endpoint that accepts a POST with data, matches the request with a model, passes the data to Postgres, and sends a response.

Overview

Here's the flow of what we will have after this module:



Don't be overwhelemed. Notice that we've added usage of our model and our database.

Code

Now that we know a POST route works, let's add in our model and our database in another controller method. This time you'll need to import the `test.js` model. You'll put the next controller method underneath the first one inside `testcontroller.js`:

```
var express = require('express')
var router = express.Router()
var sequelize = require('../db');
var TestModel = sequelize.import('../models/test'); //1

/*************************************
 * Controller Method #1: Simple Response
*************************************/
router.post('/one', function(req, res){
  res.send("Got a post request.")
});

/*************************************
```

```
 * Controller Method #2: Persisting Data
**************************************/
router.post('/two', function (req, res) {
  let testData = "Test data for endpoint two"; //2

  TestModel //3
    .create({ //4
        //6
      testdata: testData //5
    }).then(dataFromDatabase => {
        res.send("Test two went through!")
    })
});

module.exports = router;
```

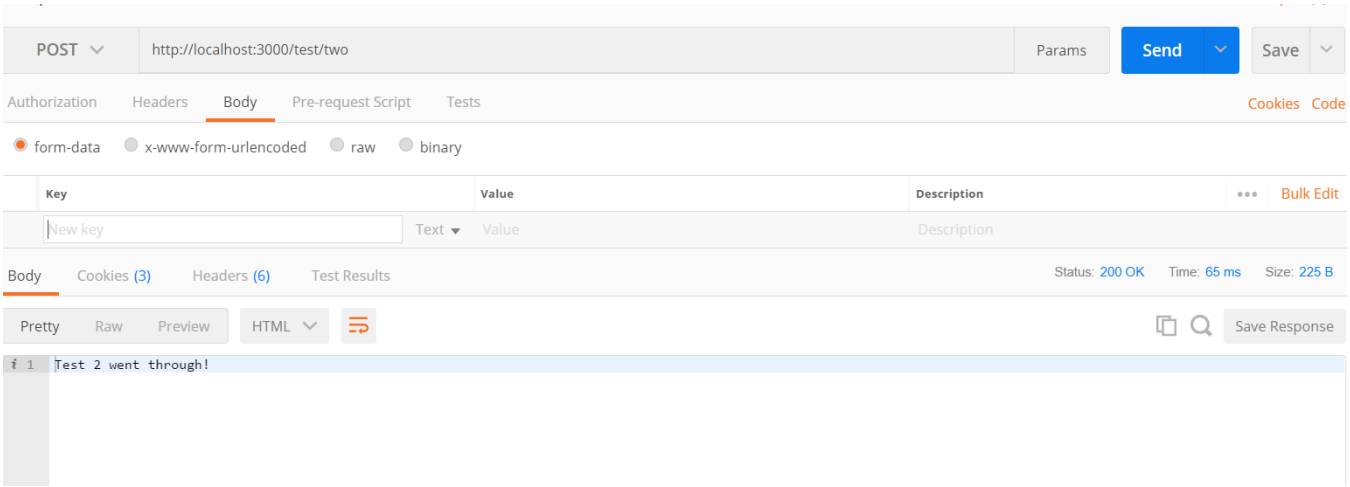
Analysis

1. We import the test model and store it in `TestModel` variable. It is convention to use Pascal casing (uppercase on both words) for a model class with Sequelize. You'll find this to be true in other programming languages as well.
2. `testData` is going to have a fixed string that we'll use every time a POST request comes in.
3. We use the `TestModel` variable to access the model that we are using. This will grant us access to the `Test` model properties and to Sequelize methods.
4. `.create()` is a Sequelize method that allows us to create an instance of the `Test` model and send it off to the db, as long as the data types match the model.
5. We pass the value of `testData` down to satisfy the key/value pair for the model. The string that we are sending will be the value that's stored in the variable. Currently, it is the string `Test data for endpoint two`;
6. `testdata` is the key in the object, and it represents the column being used in the table.

Let's test this code in Postman and do some more work that should clarify this process.

Test

1. Make sure your server is running.
2. Open Postman.
3. Open a new request.
4. Change the dropdown to POST.
5. Enter the endpoint into the URL: `http://localhost:3000/test/two`.
6. Press 'Send'.
7. You should see the following response:

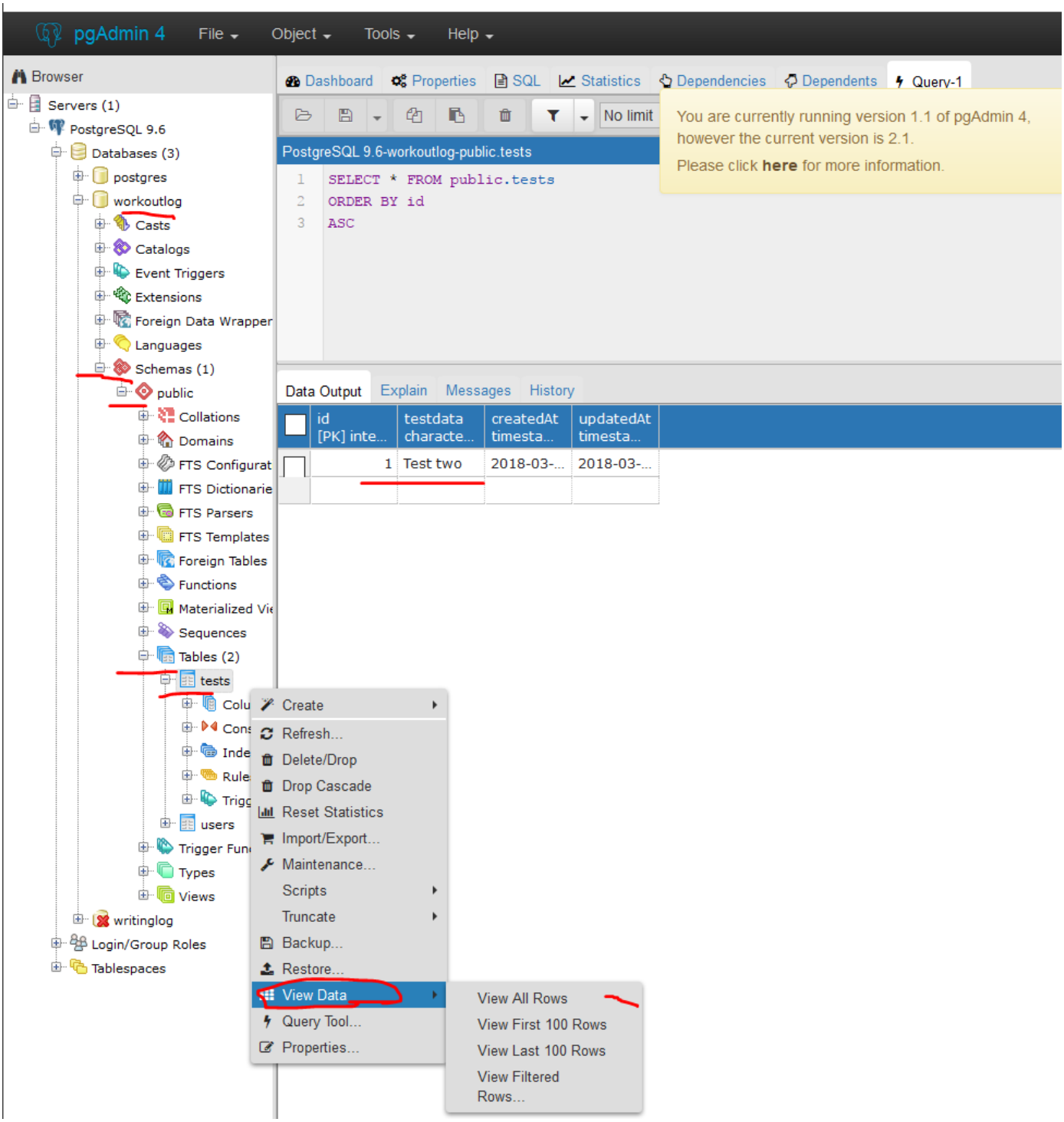


Checking the Database

Guess what? We also should have just persisted our data to Postgres. Let's check using the following steps:

1. Open PGAdmin. (The elephant)

2. Find your `workoutlog` database.
3. Find `Schemas` dropdown.
4. Navigate your way to `Tables`.
5. You should see the `tests` table.
6. Right click on it, then click `View Data` -> `View All Rows`.
7. You should see the following. Note that we have underlined all of these steps in the image:



Summary of the Flow

In this module, the following flow is happening:

1. We make a POST request with Postman.

2. The router sends that request to the `testcontroller`.
3. The `testcontroller` method contains a variable called `testData` that has a string in it.
4. Our `testcontroller` method access the `TestModel`.
5. We then use the Sequelize `create()` method to create the object to be sent to the DB.
6. The object is sent and Postgres stores it.
7. The controller sends a response to Postman.

There is a lot going on now, but we still need to iterate and add more endpoints, routes, and controller steps to see how a full controller method should be built.

req.body

03 - req.body()

In this module, we'll set up our Express application so that it can take incoming requests properly.

Overview

Hopefully, your model works, and you were able to put something in the database. That's great, but if we continue doing a POST request, we would send the same string every time. This one:

```
let testData = "Test data for endpoint two";
```

We definitely want to be more versatile with incoming requests. To do that, we need another tool called `body-parser`, and we want to use the `req.body` middleware tool provided by Express. This is confusing, so let's step through it slowly.

The Code

Go into the `testcontroller.js` file and add the following method. Add it to the bottom of the file, above the export statement:

```
/**************************************
 * Controller Method #3: req.body
**************************************/
router.post('/three', function (req, res) {
                  //1
  var testData = req.body.testdata.item;

  TestModel
    .create({ //2
      testdata: testData
    })
  res.send("Test three went through!")
  console.log("Test three went through!")
});

module.exports = router;
```

1. Here we use the `req.body` middleware provided by Express and append two more properties to it. This is what we're sending to the database. `req` is the actual request, and `body` is where our data is being held. `testdata` is a property of `body`, while `item` is a property of `testdata`. We'll see this in Postman in a little while.
2. `create()` is a Sequelize method. It creates a SQL statement that will insert our data into the database. You'll learn more about SQL later.

body-parser

In order to use the `req.body` middleware, we need to use a middleware package called `body-parser`. We installed this earlier. Go into `app.js` and add the following code:

```
var express = require('express');
var app = express();
var test = require('./controllers/testcontroller')
var sequelize = require('./db');
var bodyParser = require('body-parser'); //1

sequelize.sync(); // tip: {force: true} for resetting tables

app.use(bodyParser.json()); //2

app.use('/test', test)

app.listen(3000, function(){
```

```
    console.log('App is listening on 3000.')
});
```

Analysis

1. We pull in the `body-parser` library and store it in the `bodyParser` variable.
2. This `app.use` statement MUST go above any routes. Any routes above this statement will not be able to use the `bodyparser` library, so they will break. You should read through **this article** (https://medium.com/@adamzerner/how-bodyparser-works-247897a93b90) to get a starter understanding of how `body-parser` is working with `req.body`. Warning: this will lead you down a rabbit hole of understanding. For our purposes, it's important to know this:

> `app.use(bodyParser.json())` tells the application that we want `json` to be used as we process this request.

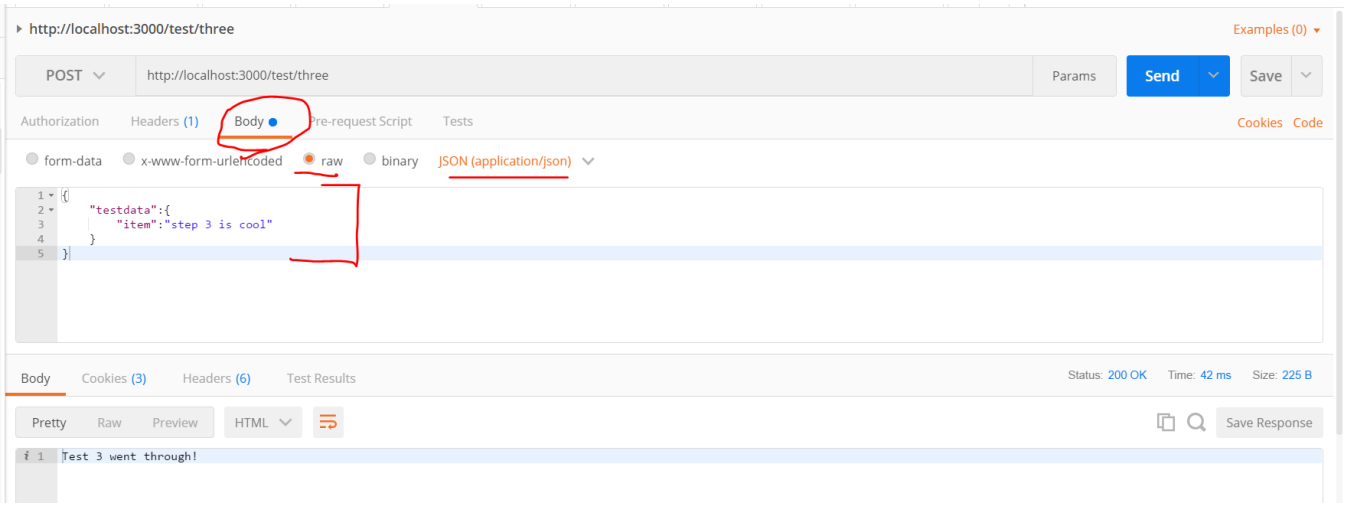
Here (https://stackoverflow.com/questions/38306569/what-does-body-parser-do-with-express) is a little more information on `body-parser`.

Testing

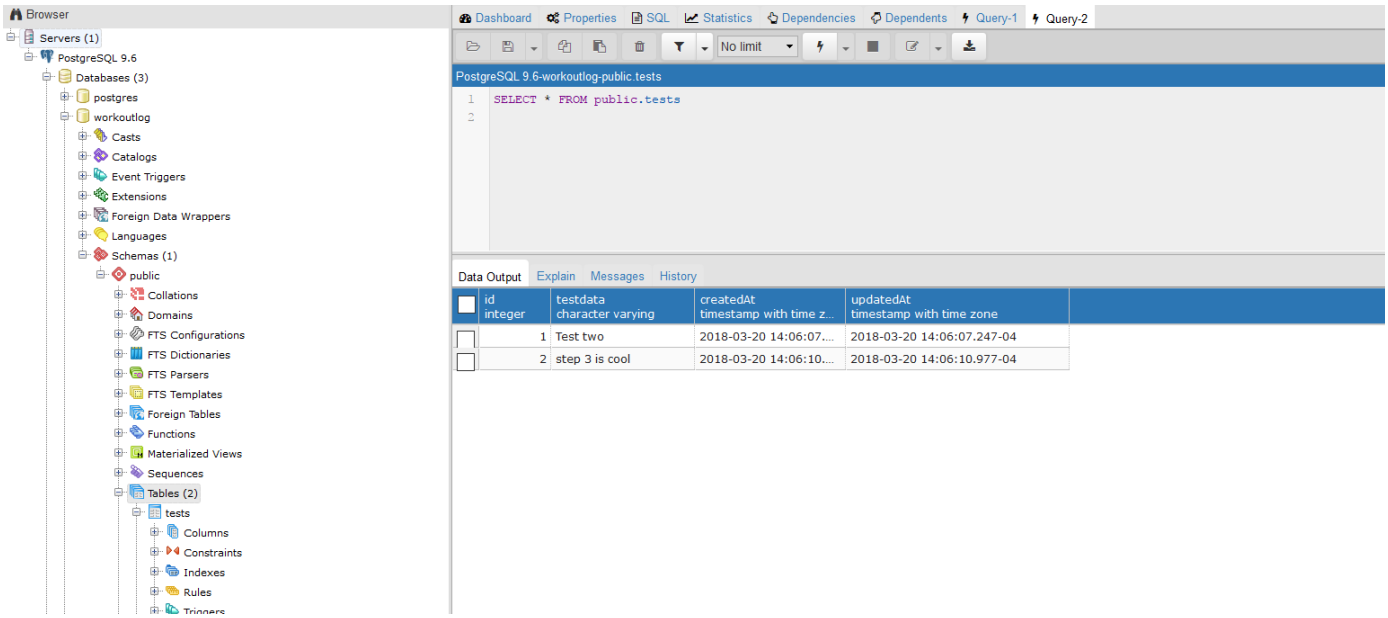
1. Make sure your server is running.
2. Open Postman.
3. Open a new request.
4. Change the dropdown to POST.
5. Enter the endpoint into the URL: `http://localhost:3000/test/three`.
6. Click on the body tab under the url input field.
7. Choose the `raw` radio button.
8. In the dropdown, choose `JSON (application/json)`.
9. In the empty space, add a JSON object like the one below. Notice that this is `testdata.item`, the last two properties in `req.body.testdata.item`. This information is inside `body`, which is inside `req`.

```
{
 "testdata":{
     "item":"step 3 is cool"
 }
}
```

10. Press send.
11. You should see the following



12. Let's also go to Postgres and make sure the data is there. To update the table, you can press the `Execute` button (the lightning bolt).



Summary of the Flow

In this module, the following flow is happening:

1. We make a POST request with Postman.
2. `body-parser` breaks the request into JSON.
3. The router sends the request to the `testcontroller`.
4. The controller with the `/three` endpoint is called.
5. The `req.body.testdata.item` is captured in the `testData` variable.
6. We then use the Sequelize `create()` method to create the object to be sent to the DB.
7. The object is sent, and Postgres stores it.
8. The controller sends a response to Postman.

Crafting the Response

04 - Crafting the Response

In this module, we'll use the `then()` function to return a Promise for our request.

Overview

It's great that our model and routes are working, but there was a slight hiccup with our last post: If you look at the console, you'll see that the success message actually printed BEFORE the data was inserted into the database. What if the insert had failed though and the data couldn't be entered? For this reason, we need to make sure that the response to the user comes AFTER the insert statement.

The Code

Go into the `testcontroller.js` file and add the following method. Add it to the bottom of the file, but above the export statement.

```
//STEP 4 - Use this with Postman
router.post('/four', function (req, res) {
  var testData = req.body.testdata.item;
  TestModel
    .create({
      testdata: testData
    })
    .then( //1
      function message() { //2
       res.send("Test 4 went through!");
      }
    );
});
```

Analysis

Here are the updates that we've made: 1. We call the `then()` method. As you'll read in the the MDN docs, the `then()` method returns a Promise. Hence, we use this asynchronous function to force the message to wait for the insert statement to finish. 2. The callback function will print the success message to the console once `testData` is done running.

Testing

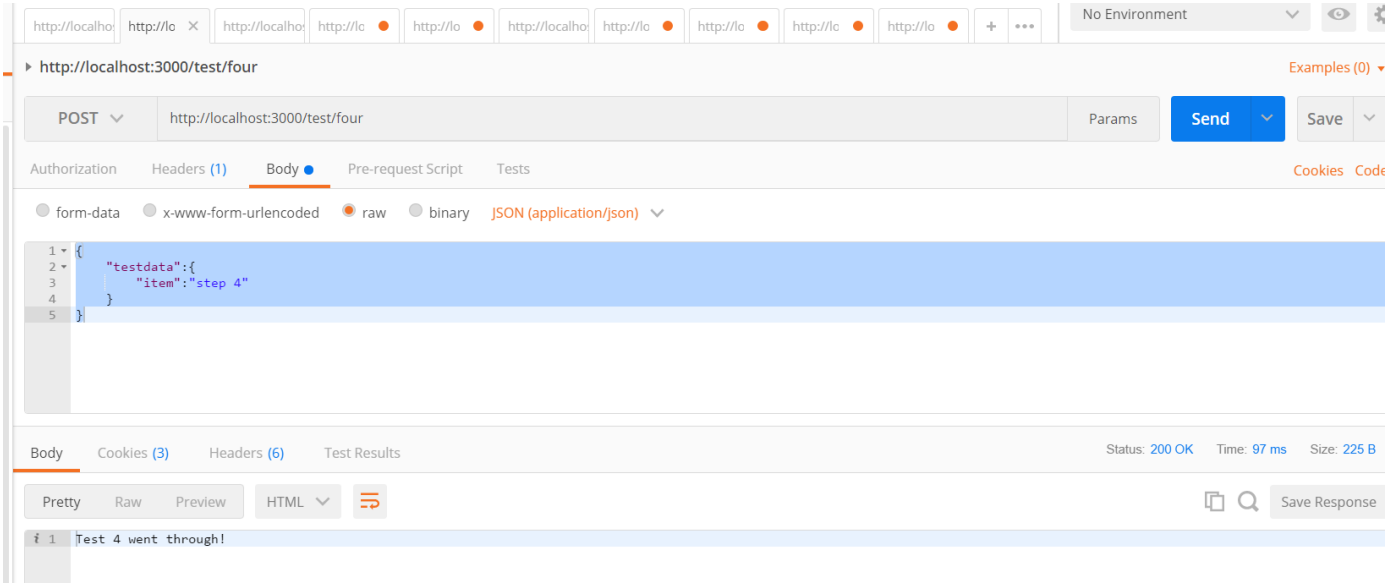
Let's use Postman to test this:

1. Make sure your server is running.
2. Open Postman.
3. Open a new request.
4. Change the dropdown to POST.
5. Enter the endpoint into the URL: `http://localhost:3000/test/four`.
6. Click on the body tab under the url input field.
7. Choose the `raw` radio button.
8. In the dropdown, choose `JSON (application/json)`.
9. In the empty space, add a JSON object like the one below:

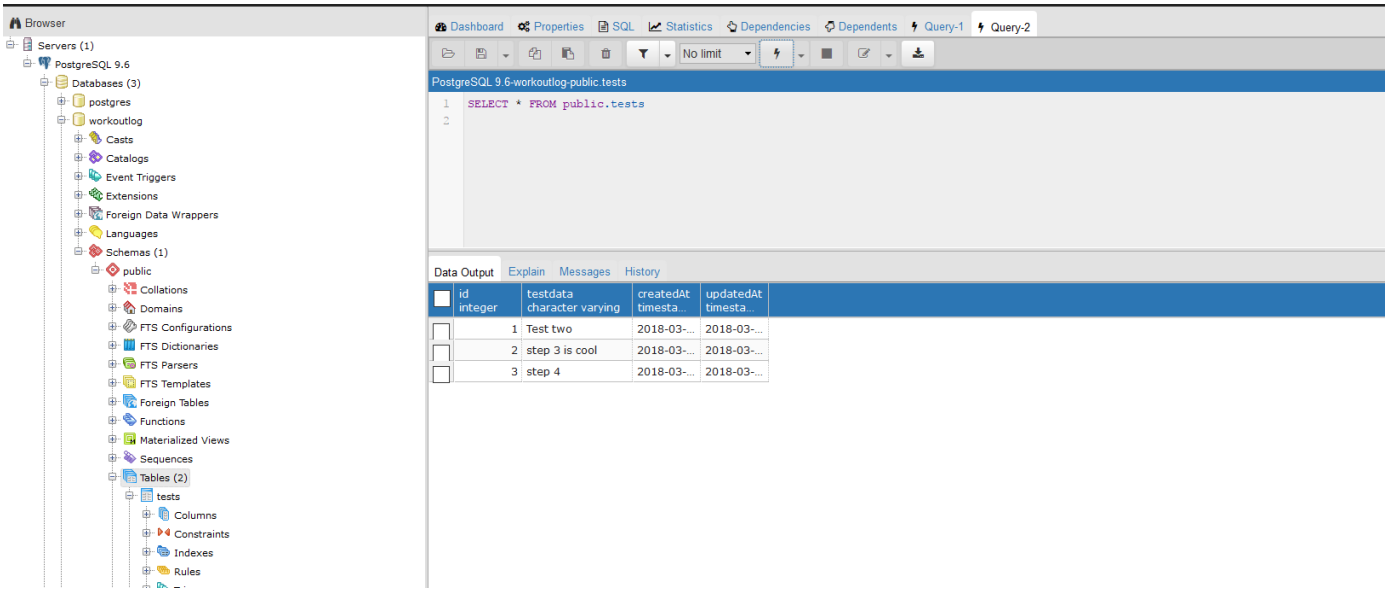
```
{
    "testdata":{
        "item":"step 4"
```

```
    }
}
```

1. Press send.
2. You should see the following:



1. Let's also go to Postgres and make sure the data is there. To update the table, you can press the `Execute` button (the lightning bolt).



Summary of the Flow

In this module, the following flow is happening:

1. We make a POST request with Postman.
2. `body-parser` breaks the request into JSON.
3. The router sends the request to the `testcontroller`.
4. The controller with the `/four` endpoint is called.
5. The `req.body.testdata.item` is captured in the `testData` variable.
6. We then use the Sequelize `create()` method to create the object to be sent to the DB.
7. The object is sent to Postgres, which stores it.

8. After the data is stored, we fire the `then()` method, which returns a Promise.
9. A method fires a response to Postman.

Sending the Response

05 - Sending the Response

In this module, we'll pass the request data back in our callback function

Overview

We now have a proper sequence, but the way we displayed the success message might not be what we want. We actually are probably going to want the data to be sent back to us after it's been saved in the db. The current response `res.send("Step four");` is in no way really connected to the actual data. What if, for instance, we were saving an item to a To-Do list? We would want the data to save, and then we would want to get it in our response. Let's pass some of the data into the callback to provide a more detailed response to the user.

The Code

Go into the `testcontroller.js` file and add the following method. Add it to the bottom of the file above the export statement.

```
/*********************
 * Route 5: Return data in a Promise
 *********************/
router.post('/five', function (req, res) {
  var testData = req.body.testdata.item;
  TestModel
    .create({
      testdata: testData
    })
    .then(              //1
      function message(data) {
        res.send(data);  //2
      }
    );
});
```

1. It's important to note that the `.then()` method is chained to `.create()`. In fact, the whole expression could be read like this in one line:

   ```
   TestModel.create({testdata: testData}).then(function message(data) { res.send(data);});
   ```

2. With that idea in mind, we have changed the data coming back in the response to the data that was persisted in Postgres. To be clear, after the data is persisted in the Postgres with the `.create()` method and in the `testdata` column, the `.then()` method returns a Promise that fires up a callback function holding the data that was just added.

It's important to note that the `data` parameter can have any name that we want.

Testing

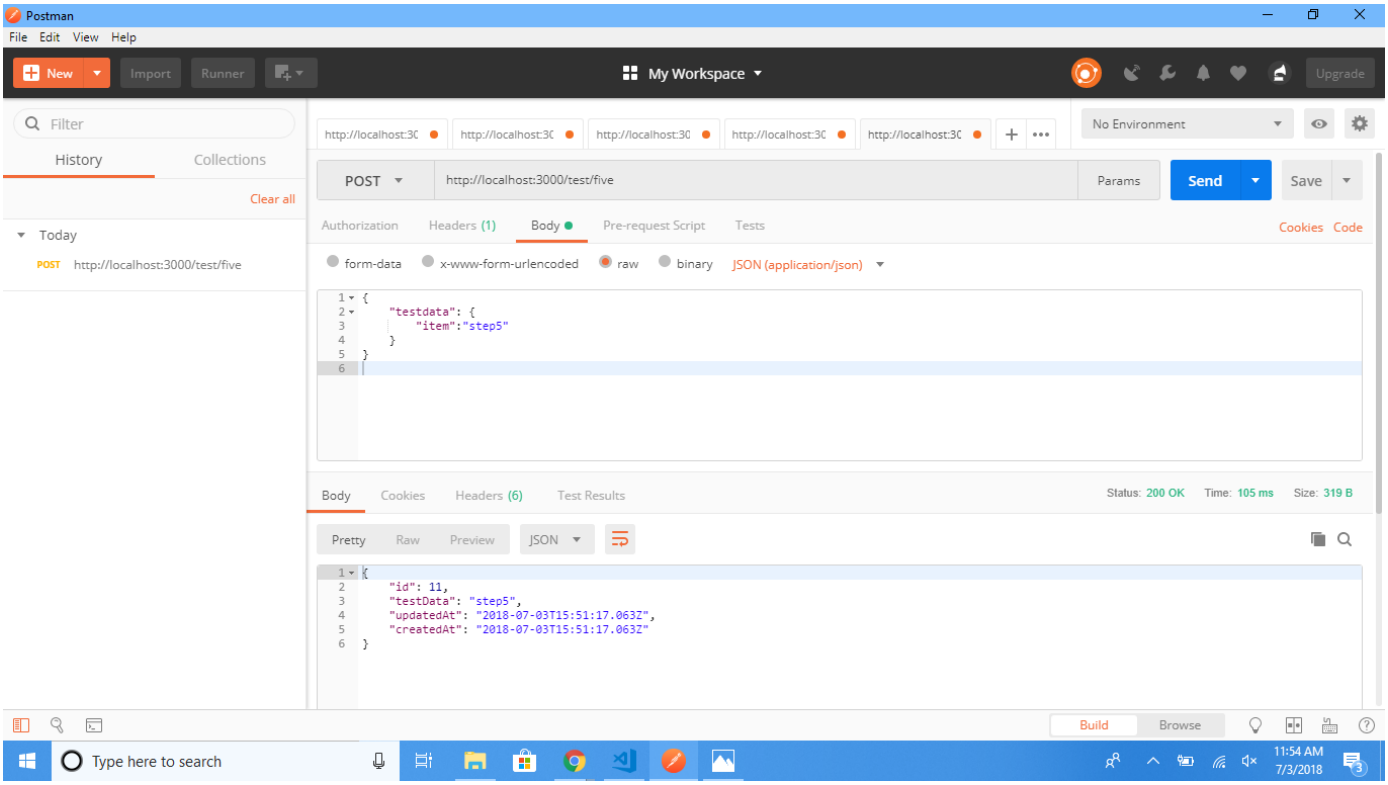
Let's use Postman to test this:

1. Make sure your server is running.
2. Open Postman.
3. Open a new request.
4. Change the dropdown to POST.
5. Enter the endpoint into the URL: `http://localhost:3000/test/five`.
6. Click on the body tab under the url input field.

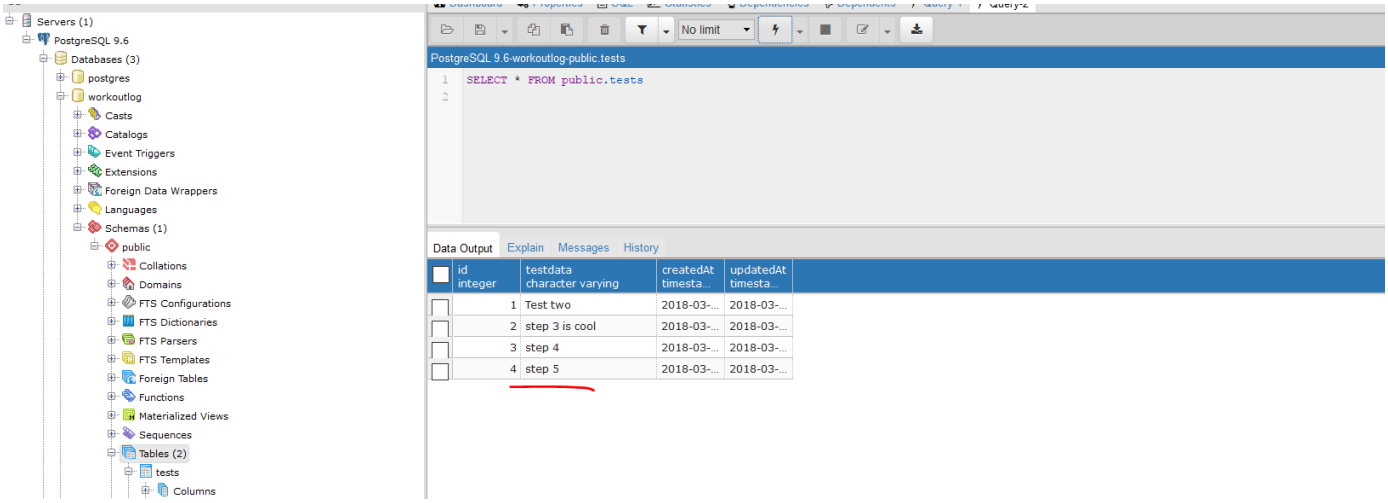
7. Choose the `raw` radio button.
8. In the dropdown, choose `JSON (application/json)`.
9. In the empty space, add a JSON object like the one below:

```
{
    "testdata":{
        "item":"step 5"
    }
}
```

1. Press send.
2. You should see the following:



1. Notice that the data in the response matches the data in the request.
2. You should also go to Postgres and make sure the data is there and that the `testdata` column matches the request and response:



Summary of the Flow

In this module, the following flow is happening:

1. We make a POST request with Postman.
2. `body-parser` breaks the request into JSON.
3. The router sends the request to the `testcontroller`.
4. The controller with the `/five` endpoint is called.
5. The `req.body.testdata.item` is captured in the `testData` variable.
6. We then use the Sequelize `create()` method to create the object to be sent to the DB.
7. The object is sent to Postgres, which stores it.
8. After the data is stored, we fire the `then()` method, which returns a Promise.
9. We call a method that takes in a parameter called `testdata`. It holds the response data.
10. The method fires a response to Postman.

Json Response

06 - JSON Response

In this module, we'll return the response as JSON instead of a simple string.

Overview

We now have a proper sequence and a response with some of the stored data, but what if we want to know when the data was stored? We have that in the information in the database, but at the moment it is not coming back in the response. How do we do that? Let's look.

The Code

Go into the `testcontroller.js` file and add the following method. Add it to the bottom of the file above the export statement.

```
/*********************
 * Route 6: Return response as JSON
 *********************/
router.post('/six', function (req, res) {
  var testData = req.body.testdata.item;
  TestModel
    .create({
      testdata: testData
    })
    .then(
      function message(testdata) {
        res.json({ //1
          testdata: testdata  //2
        });
      }
    );
});
```

Analysis

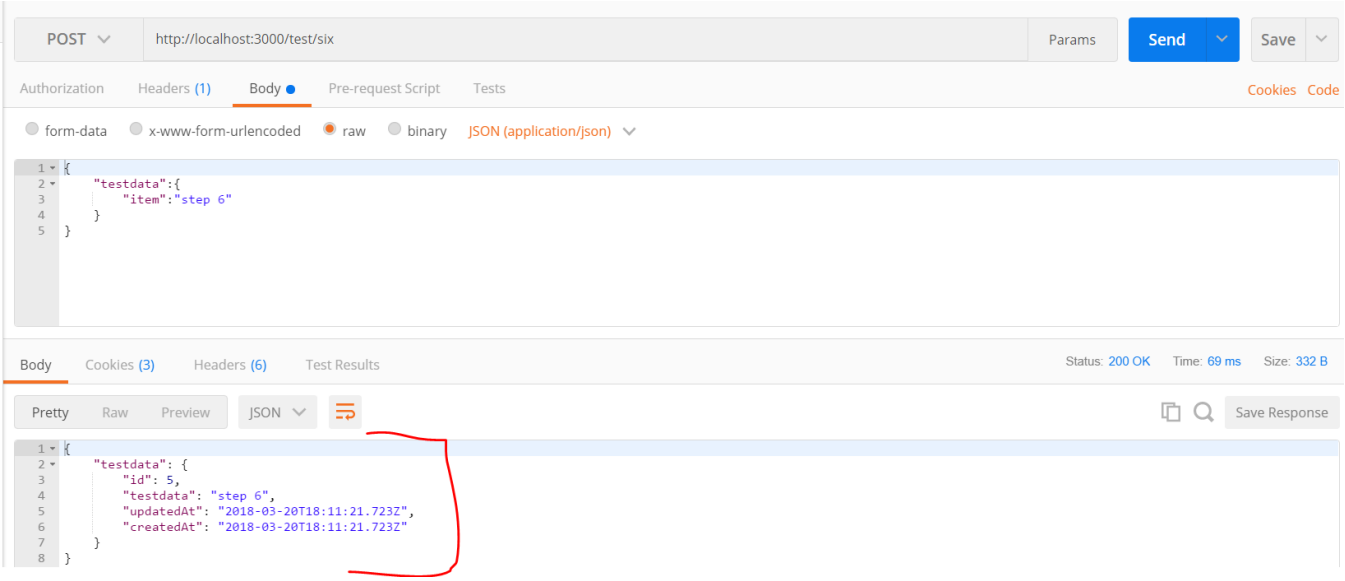
1. In our callback, rather than `res.send()`, we will invoke the `.json()` method. This will, of course, package our response as `json`.
2. The same object that was added to the database is now being sent back to the client and stored in a `testdata` property.

Testing

Let's use Postman to test this: 1. Make sure your server is running. 2. Open Postman. 3. Open a new request. 4. Change the dropdown to POST. 5. Enter the endpoint into the URL: `http://localhost:3000/test/six`. 6. Click on the body tab under the url input field. 7. Choose the `raw` radio button. 8. In the dropdown, choose `JSON (application/json)`. 9. In the empty space add, a JSON object like the one below:

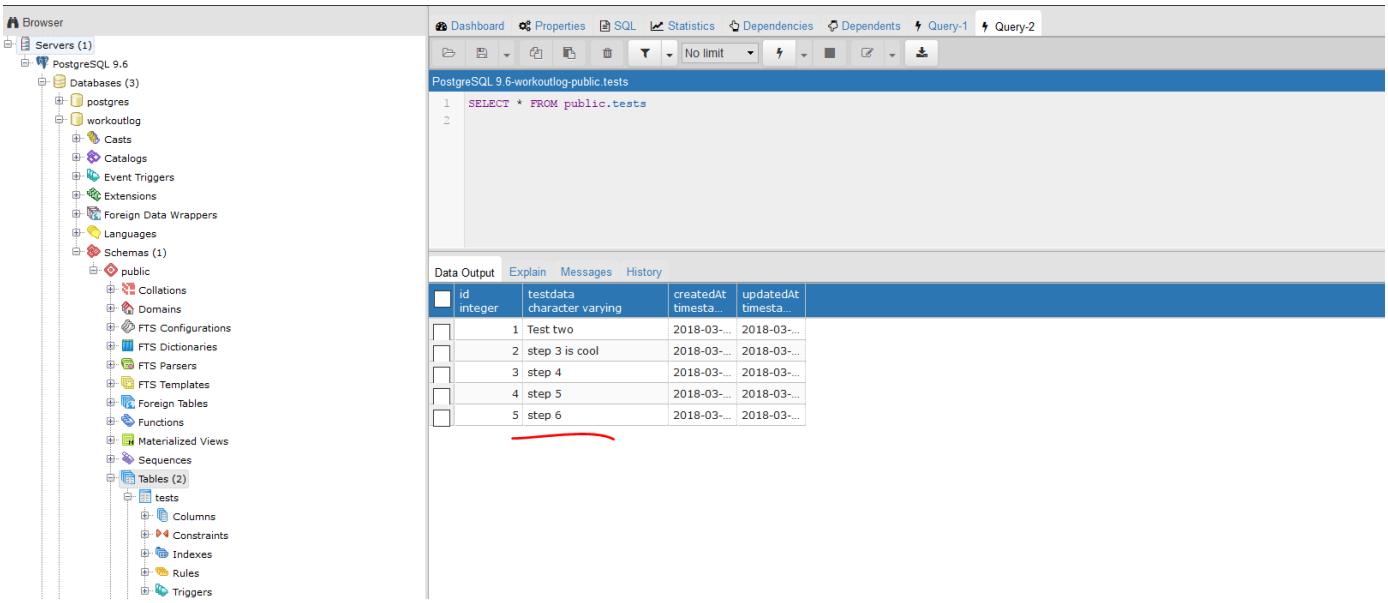
```
{
    "testdata":{
        "item":"step 6"
    }
}
```

1. Press send.
2. You should see the following:



3. Notice that the data in the response matches the data in the request, but we are also getting back a full JSON object from the database, including timestamp data.

4. You should also go to Postgres and make sure that the data is there and that the `testdata` column matches the request and response:



Summary of the Flow

In this module, the following flow is happening:

1. We make a POST request with Postman.
2. `body-parser` breaks the request into JSON.
3. The router sends the request to the `testcontroller`.
4. The controller with the `/six` endpoint is called.
5. The `req.body.testdata.item` is captured in the `testData` variable.
6. We then use the Sequelize `create()` method to create the object to be sent to the DB.
7. The object is sent and Postgres stores it.
8. After the data is stored, we fire the `then()` method, which returns a Promise.

9. We call a method that takes in a parameter called `testdata`. It holds the data for the response.
10. The method sends the data back as JSON this time, and the response goes to Postman.

Error Handling

07 - Error Handling

In this module we'll write code that would handle errors if something in our server were to go wrong.

Overview

What about the idea of a network problem as we're making our request? We make the request, the network drops, and the response never comes. What if something is wrong with our server code? We're going to need to create a function that handles such errors.

The Code

Go into the `testcontroller.js` file and add the following method. Add it to the bottom of the file, above the export statement.

```
/*********************
 * Route 7: Handle errors
 **********************/
router.post('/seven', function (req, res) {
  var testData = req.body.testdata.item;

  TestModel
    .create({
      testdata: testData
    })
    .then(
      function createSuccess(testdata) {
        res.json({
          testdata: testdata
        });

      },
      function createError(err) { //1
        res.send(500, err.message);
      }
    );
});

module.exports = router;
```

Analysis

1. The addition that we've made here is an error function. If the `create()` function returns an error, it will be picked up by the `createError()` method. That method will then send back a `500` error with a message.

Summary of the Flow

In this module, the following flow is happening:

1. We make a POST request with Postman.
2. `body-parser` breaks the request into JSON.
3. The router sends the request to the `testcontroller`.
4. The controller with the `/seven` endpoint is called.
5. The `req.body.testdata.item` is captured in the `testData` variable.
6. We then use the Sequelize `create()` method to create the object to be sent to the DB.
7. The object is sent and Postgres stores it.

8. After the data is stored, we fire the `then()` method, which returns a Promise.
9. We call a method that takes in a parameter called `testdata`. It holds the data for the response.
    1. The method sends the data back as JSON this time, and the response goes to Postman.
    2. We also have an error for cases when our server throws an error.

Conclusion

04 - Conclusion

In this module, we'll take a look back at what we've done.

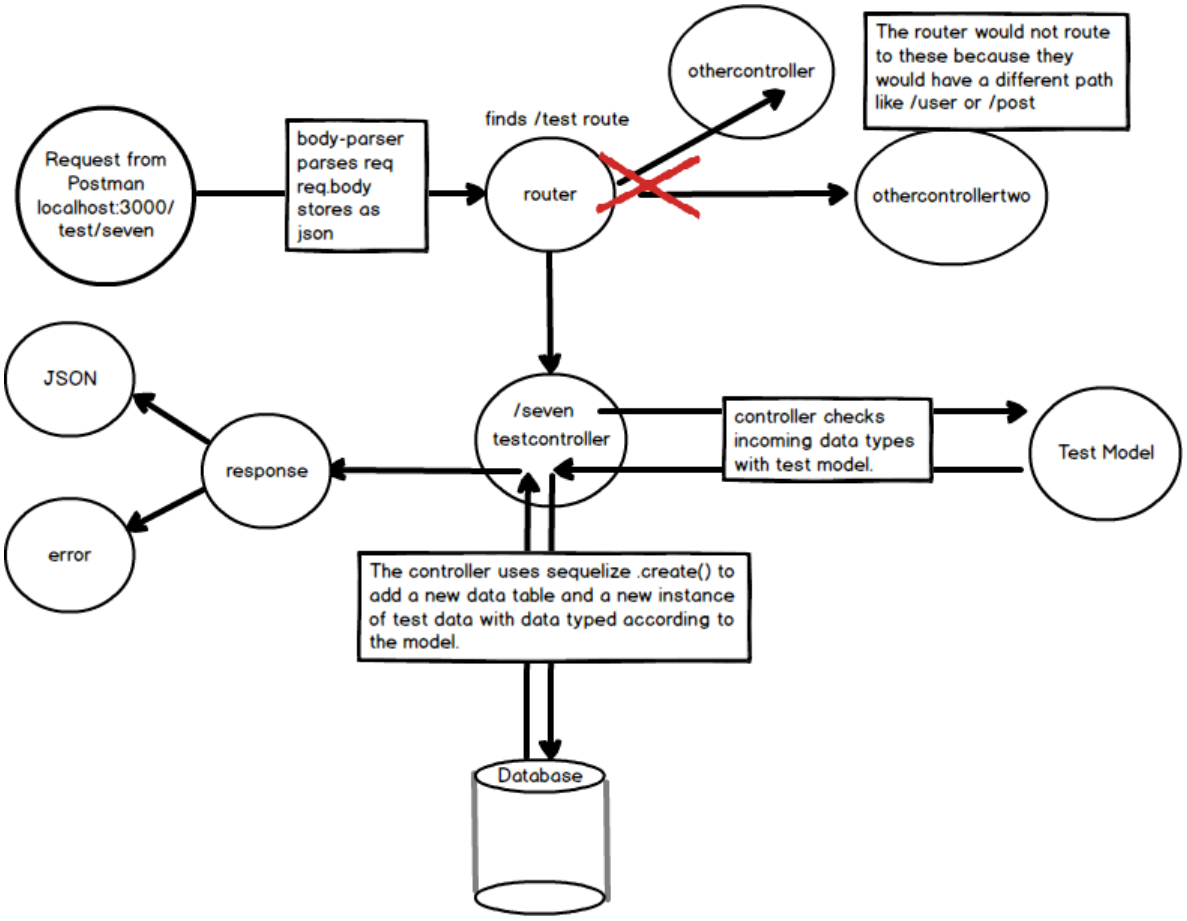
Overview

In Section 5, we've done the following:

1. A step-by-step creation of a POST controller method.
2. Learned about passing data to our database.
3. Gained experience with using Postman to test HTTP requests and responses.
4. Begun working with the Model-View-Controller software pattern.
5. Learned more about Promises and used them in a practical way.
6. Worked with Sequelize methods and models to help create data to be persisted.

Diagram

Let's take a look at the big picture diagram:



Moving On

This chapter will be the foundation for a pattern that we will continually use as we build out our server side. A good way to learn to code is to work through bite-sized chunks, referencing the documentation, and playing with test data, all of which we have been doing. Although a good amount of heavy lifting has been done, we have only just begun. As the controller method that we wrote in seven steps becomes second nature to you, you'll see that learning is a continuous requirement in our field.

JWT Intro

01 - JWT intro

In this module, we'll introduce JSON Web Token (JWT), sometimes referred to as Jot.

What is JWT?

JWT stands for **J**SON **W**eb **T**oken. It's an identifier to be added to the body of a request, so that when you send a request, you send a token with it. Let's look at a good analogy for how JWT works:

You go to a venue to see a band. The venue is charging a $15 entry fee. You pay the fee, and then the doorman stamps your hand. You go in and the show starts. Think of this as like signing up for an app, and your stamp is the token.

You leave the venue to get pizza and talk to your date. When you go back in, the doorman, who doesn't remember you at all, looks at your hand to see the stamp. When he sees it, he lets you in. That is what happens when you make another request in an app.

This is a real-world version of using JWT. You are the client, the venue is the server, and the stamp on your hand is the token. The doorman won't let you in without paying or a stamp. The app won't let you in without signing up or having a token with your request. And just like a stamp that will wash away in a few days, the JSON token will wear off in a certain amount of time. This time is decided by the developer.

Statelessness Review

The token deals with the issue of statelessness in HTTP. Think back to the idea of statelessness. Once a request is sent and a response is returned, both sides forget that the other side exists. The client forgets the server, and the server forgets the client. Once you walk into the music venue, the doorman forgets you. If you come back out, you can't get back in without the stamp. This is statelessness.

Authenticated Request

We use a token for authenticated requests; which are requests for which a user has to be authenticated (logged in) to do CRUD type stuff:

- A user saves a note.
- A user looks at a friends tweet.
- A user direct messages a friend.
- A user loads up a binge-worthy Netflix show on their account, and a record of that is stored in their parents' account because they are still free-sharing.

These requests require tokens if the user is logged in. They are authenticated requests.

Token vs. Cookie

It's important to know about the difference between a token and a cookie, and it's drifting out side of our scope right now. We will say that we are using token-based authentication. However, the cookie and token talk is necessary reading for you. Here's a required **article (https://auth0.com/blog/cookies-vs-tokens-definitive-guide/)** , and a **Stack Overflow (https://stackoverflow.com/questions/17000835/token-authentication-vs-cookies)** post.

User Create

01 - User Create

In this module, we'll start to set up the necessary items for creating a new user.

So far, we've just been putting data into a table in our database. However, if someone were to look at that data, there's no way to tell who actually put the data there. Additionally, there's no security protocols in place, so anyone can connect to the database and modify its contents in whatever way they choose. We can use JWT to fix both of these issues. We have a little bit of prep to do first, though: We'll use Sequelize and create a User model to create a new user in the database.

User Model

Let's set up a new user model. Create a file `user.js` in the `models` folder.

```
module.exports = function (sequelize, DataTypes) {
          //1      //2
    return sequelize.define('user', {
        username: DataTypes.STRING, //3
        passwordhash: DataTypes.STRING //3
    });
};
```

Analysis

This should look familiar:

1. A function with a Sequelize object that calls the `define` method.
2. A first parameter that will create a `users` table in Postgres.
3. An object with `username` and `passwordhash` that will be the columns in the table. We'll talk more about a `passwordhash` later.

app.js

We'll need to set up a route to the user controller methods in `app.js`. We have added the entire file for orientation:

```
var express = require('express');
var app = express();
var test= require('./controllers/testcontroller')
var user = require('./controllers/usercontroller') //1
var sequelize = require('./db');
var bodyParser = require('body-parser');

sequelize.sync(); // tip: {force: true} for resetting tables

app.use(bodyParser.json());

app.use('/test', test);

app.use('/api/user', user); //2

//3 You could also write it this way without the require statement above.
//app.use('/api/user', require('./controllers/usercontrollers'));

app.listen(3000, function(){
    console.log('App is listening on 3000.')
});
```

Analysis

1. We import the `usercontroller.js` file.
2. We set up a route to the endpoints for the `api/user` route.
3. Just another way to write out your routes. Just be consistent.

usercontroller.js

If you haven't already, create a new file inside the `controllers` folder and call it `usercontroler.js`. We'll need to add some code to that file. Note: if Lebowski isn't your style, enter in your own flavor for the string values:

```
var express = require('express')
var router = express.Router()      //1
var sequelize = require('../db');
var User = sequelize.import('../models/user');

/************************
** Create User Endpoint: Starter***
************************/
//2
router.post('/createuser', function (req, res) {

  var username = "The Dude";
  var pass = "therugtiedtheroomtogether";               /**3**/

  User.create({
    username: username,
    passwordhash: pass

  }).then(
    function message(){
      res.send("I hate The Eagles, man");
    }
  );
})

module.exports = router;
```

Analysis

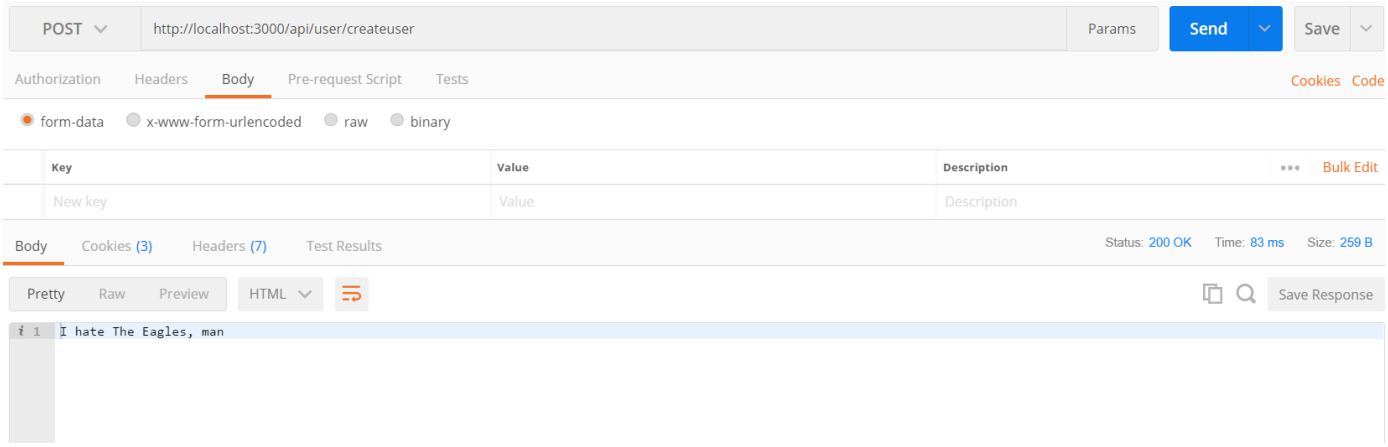
This should look familiar again:

1. We bring in our necessary imports. Same as the testcontroller, just with a User model now.
2. We start out our `POST` method for a `createuser` endpoint.
3. Inside the method, we have the basics for creating a new user and returning a message in the response.

Postman

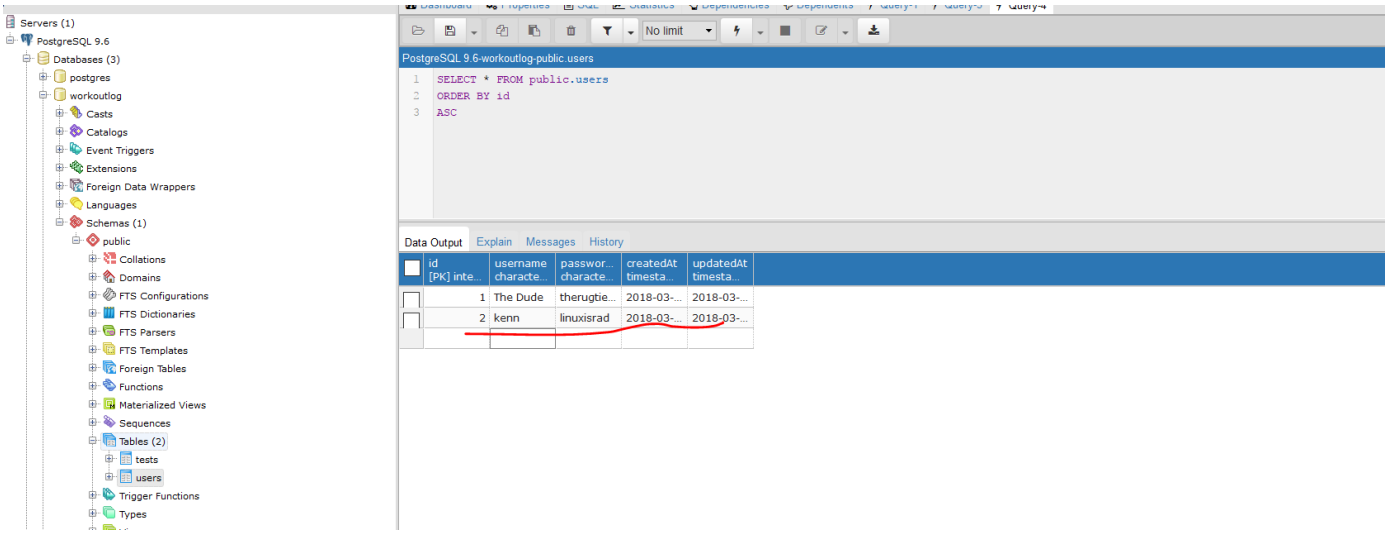
Let's quickly test this with Postman.

1. Start your server then open Postman.
2. Figure out the endpoint to send a `POST` request to.
3. Press send.
4. You should see the response string:



Postgres

You should also go check Postgres to see that the data showed up:



Refactor

02 - Refactor

In this module, we'll refactor the user create controller method.

Refactor

Refactoring is just part of the game. We work in iterations. You write methods, you add to them, you take things away. In this chapter, you won't be rebuilding a POST controller method in seven steps like we did last time. Let's do a quick refactor:

```
router.post('/createuser', function (req, res) {

  var username = req.body.user.username;
  var pass = req.body.user.password;

  User.create({
    username: username,
    passwordhash: pass

  }).then(
    function createSuccess(user) {
      res.json({
        user: user,
        message: 'created' //1
      });
    },
    function createError(err) {
      res.send(500, err.message);
    }
  );
});

module.exports = router;
```

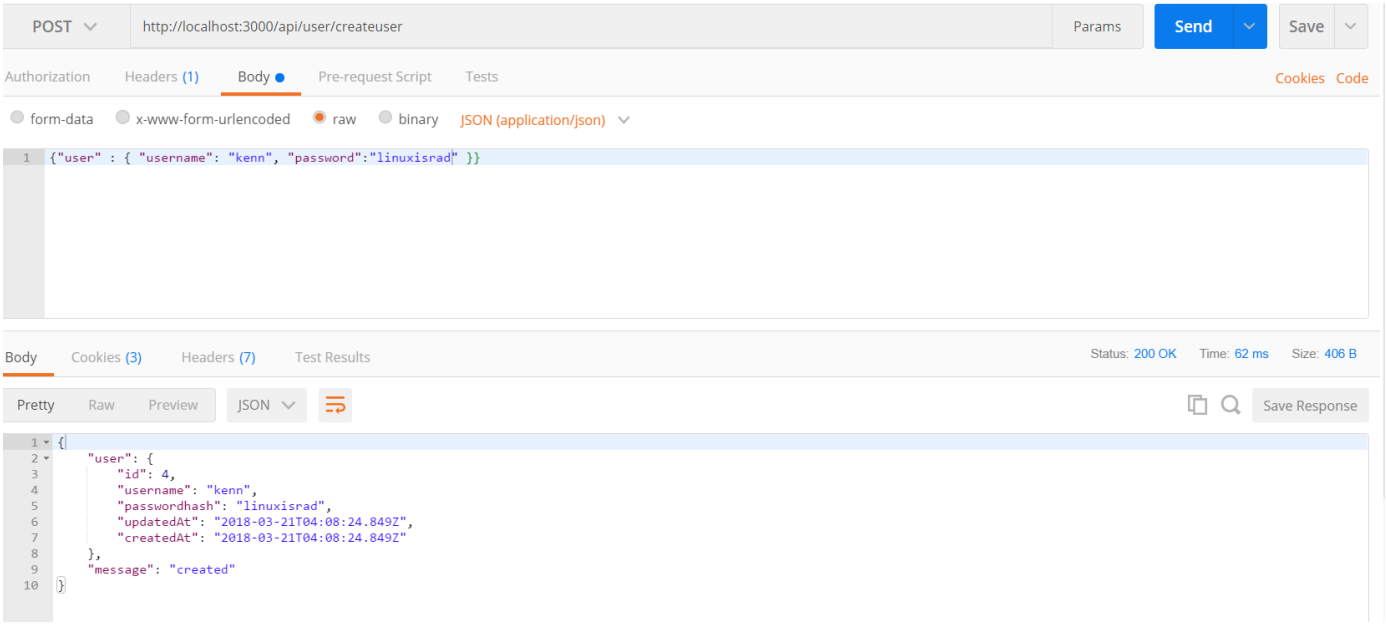
Analysis

1. Along with the user object that gets returned as JSON, we can send a message in the response.
2. For the sake of time, we'll ask you to reread and review the flow of the above method. If you don't have an understanding, you'll want to review the information in the testcontroller in the `test/seven` method. One big difference here is that we have two properties instead of one.

Postman

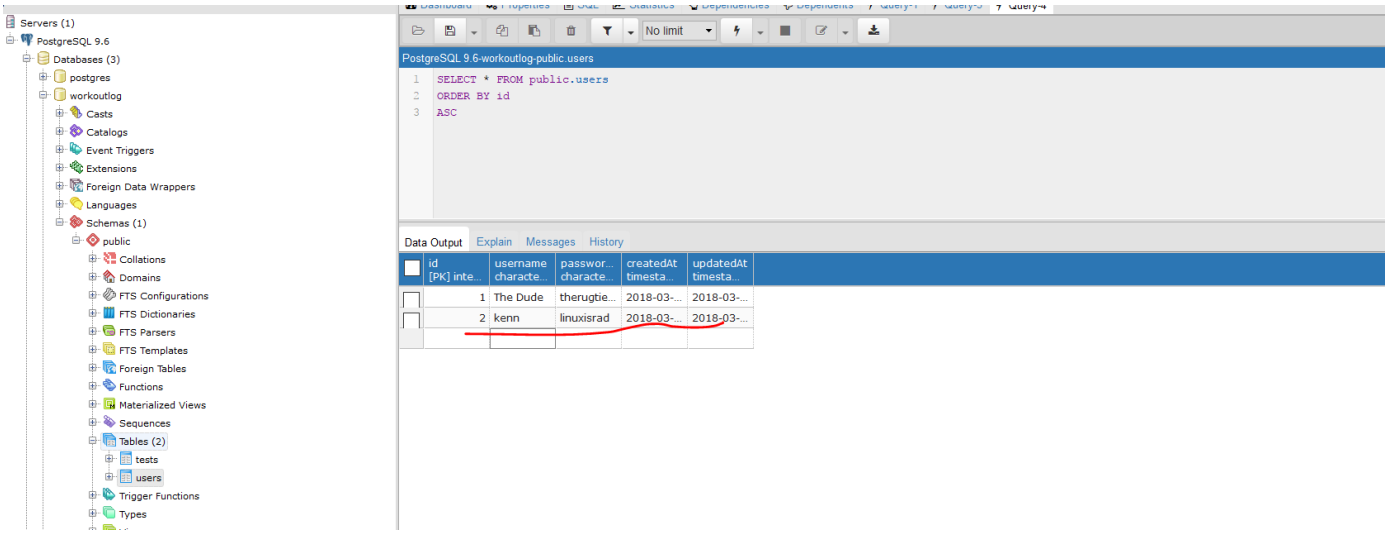
Let's quickly test this iteration of the method with Postman.

1. Start your server then open Postman.
2. Figure out the endpoint to send a post request to.
3. Go to the body tab -> Choose Raw -> Change the dropdown to JSON.
4. Enter the request body: `{"user" : { "username": "kenn", "password":"linuxsirad" }}`.
5. You should see the response string:



Postgres

Check Postgres, too:



Next

In the next module, we'll add a token to the response.

JWT Package

01 - JWT Package

We have successfully set up user accounts in our database. Until we set up authentication, however, it doesn't really do us much good. We previously installed `jwt` when you originally made your package.json file, but now we'll get it set up for use within our database.

Adding JWT

Check your `package.json`. You should have the `jsonwebtoken` package installed earlier.

Let's now go into the `usercontroller.js` file and import the package:

```
var express = require('express')
var router = express.Router()
var sequelize = require('../db');
var User = sequelize.import('../models/user');
var jwt = require('jsonwebtoken'); //<-- ADD THIS LINE
```

Adding JWT

02 - Adding JWT

TOKEN CREATION

In this module, we'll discuss tokens and start creating tokens in our app.

Token Parts

A token consists of three parts:

1. The `header` (consists of the type of token and an algorithm to encode/decode).
2. The `payload` (the data being sent via the token; in our case, the username and password).
3. A `signature` (used by the algorithm to encode/decode the token. Without the signature, the token is useless).

You don't have to dive deep now, maybe on your next iteration of learning, but you should at least go to the official **JWT (https://jwt.io/introduction/)** website and look around. You can even create your own simulated token there to see how it works. The big takeaways are that there are heavy algorithms that make the token for you and that there are different parts of the token to learn about.

The Code

Let's go inside of the `usercontroller.js` file and into the `/createuser` POST method. Add the following code to the method:

```
router.post('/createuser', function (req, res) {

  var username = req.body.user.username;
  var pass = req.body.user.password;

  User.create({
    username: username,
    passwordhash: pass

  }).then(

    function createSuccess(user) {
        //1           //2     //3           //4               //5
      var token = jwt.sign({id: user.id}, "i_am_secret", {expiresIn: 60*60*24});

      res.json({
        user: user,
        message: 'created',
        sessionToken: token //6
      });
    },
    function createError(err) {
      res.send(500, err.message);
    }
  );
});

module.exports = router;
```

ANALYSIS

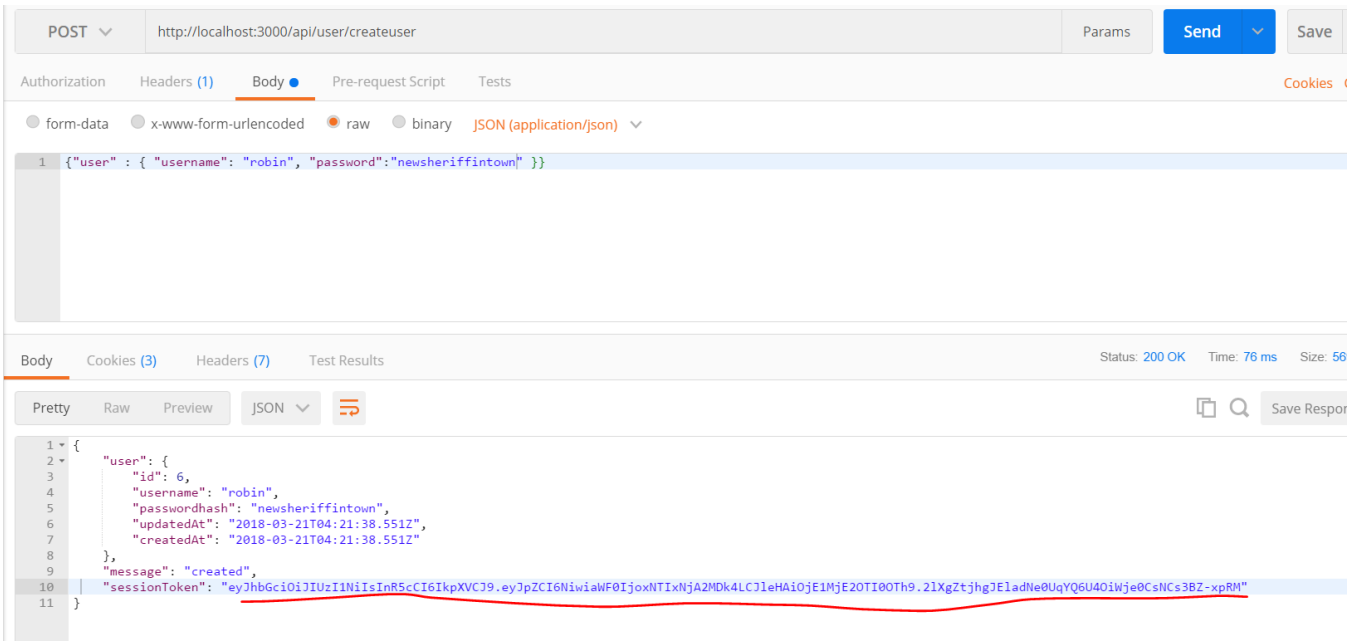
Let's walk through what we've added:

1. Create a variable to hold the token.
2. `.sign()` creates the token. It takes at least 2 parameters: the payload and the signature. You can also supply some specific options or a callback.

3. This is the payload, or data we're sending. `user.id` is the primary key of the user table and is the number assigned to the user when created in the database.
4. This is the signature, which is used to help encode and decode the token. You can make it anything you want, and we will make this private later.
5. We set an option to make the token expire. Here, we're taking (seconds *minutes* hours); in other words, 1 day.
6. We pass the value of the token back in our response. The server has now assigned a token to a specific user, and the client will have that token to work with (once we have a client).

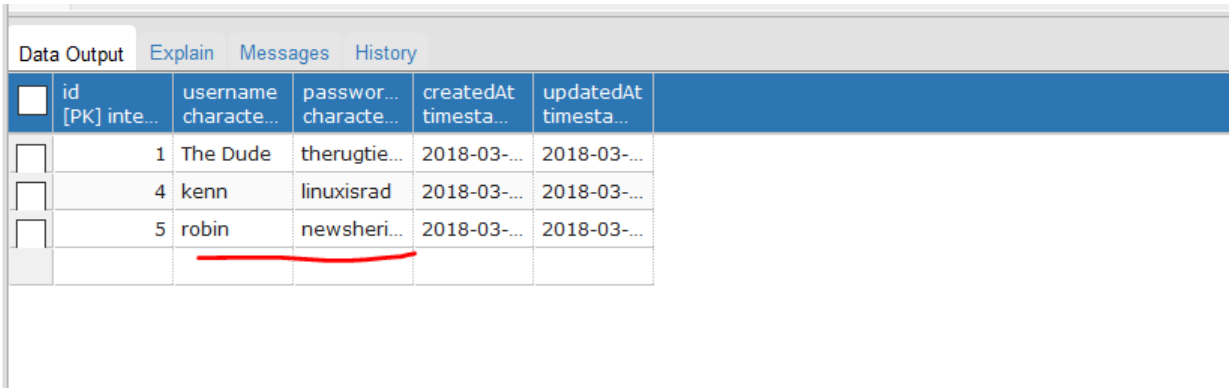
TESTING

1. Run in Postman:



   Notice that the token has come back as part of the response:

2. You should also check Postgres to be sure that the user has been added:



Review

1. When we create a user, the server will also create a token to send to the client. The server sends the token back to the client in the response. Most of the time, the client will store the token in `localStorage`, where it can be used in future requests. The token will be valid until it is removed or expired.

MAJOR SECURITY RISK

At the moment, our signature, `"i_am_secret"`, is available for everyone in the world to see via GitHub. This is extremely dangerous because there are robots and processes that scour public repositories looking for passwords and secret phrases. In our next module,

we'll discuss a way to help keep sensitive information like this hidden.

Env

03 - ENV

In this module, we'll work on making our signature private with the `.env` file.

Overview

As mentioned before, our signature is currently available to anyone who wants it on GitHub. We can use a package called `dotenv` to hold data that we want hidden, then we can have the program reach out to that file when the data is needed. We can then prevent this file from being uploaded to GitHub. `dotenv` provides a way to allow you to create secret keys that your application needs to function and keep them from going public.

File Set up

Let's start by adding a `.env` file to the root level:

```
javascript-library
    └── 5-Express Server
        └── Server
            └── controllers
            └── middleware
            └── models
            └── .env
            └── app.js
            └── db.js
```

Import dotenv Package

We have already installed the `dotenv` package. In order to use it we need to go to `app.js` and require it at the top of the file:

```
require('dotenv').config();//1 <--- ADD THIS LINE

var express = require('express');
var app = express();
var test= require('./controllers/testcontroller')
var user = require('./controllers/usercontroller')
var sequelize = require('./db');
var bodyParser = require('body-parser');
```

1. With this we can make items in an `.env` file available to our whole application.

.env File

1. Add `*.env` to your `.gitignore` to prevent it from being published to GitHub.
2. In the `.env` file, add the secret. Put it in exactly like this:

```
JWT_SECRET="i_am_secret"
```

Adding the Process Variable

Now, let's add the `process.env` variable to our method. See the comment below:

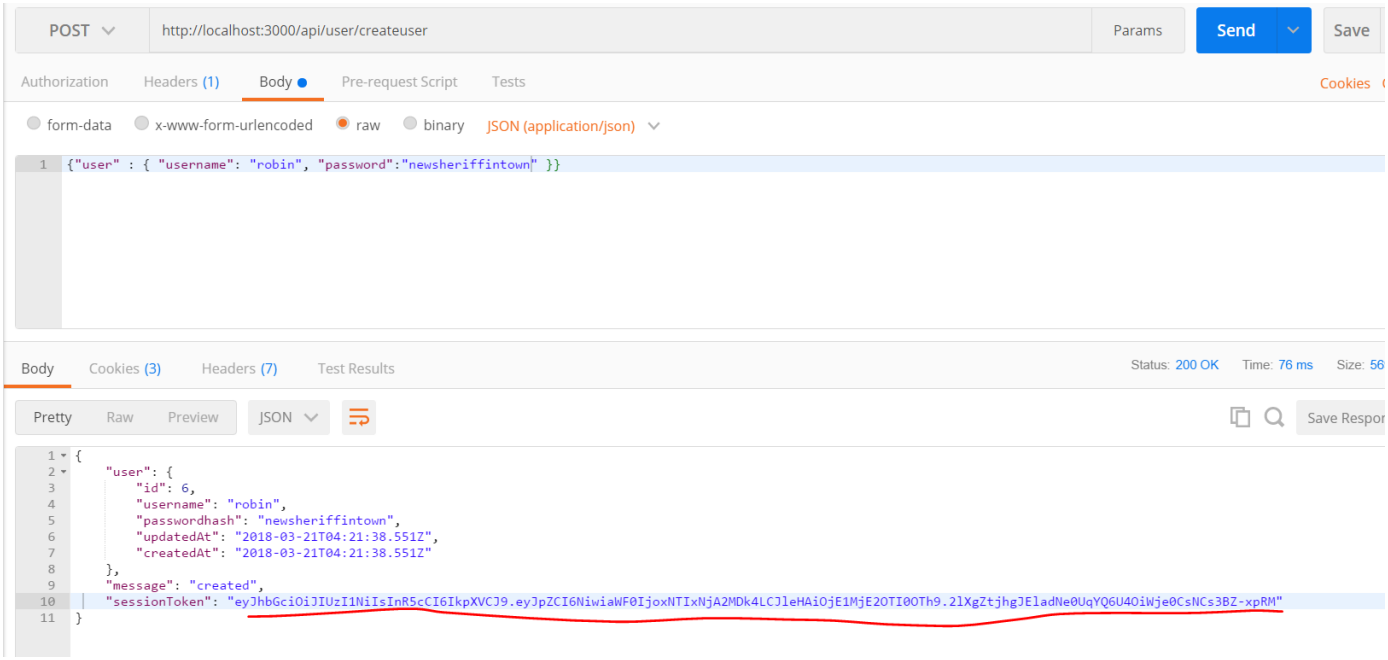
```
router.post('/createuser', function(req, res) {
        var username = req.body.user.username;
        var pass = req.body.user.password;

        User.create({
            username: username,
```

```
            passwordhash: pass
        }).then(

            function createSuccess(user){                    //1
                var token = jwt.sign({id: user.id}, process.env.JWT_SECRET, {expiresIn: 60*60*24});
                res.json({
                        user: user,
                        message: 'created',
                        sessionToken: token
                });
            },
            function createError(err){
                res.send(500, err.message);
            }
        );
    });

module.exports = router;
```

1. The system goes outside the current file to the `.env` file, where it looks for something called `JWT_SECRET`. The value of the secret is stored in that environment variable.

Test

We'll leave it up to you to test the app again with Postman and be sure that you get a token back. You should be getting the same result as you got in the last module:



bcrypt

01 - bcrypt

In this module, we'll work on storing a hashed password in the database.

Overview

As mentioned before, you never want a plain-text password returned in a response, and you never want a plain text password stored in the database. We want to hide the true value of the password, which we do through encryption.

What is Encryption?

If you've ever used a decoder ring, you've had experience with encryption. If not, here's a short video to help explain it (short into; encryption begins at 36 seconds):

Think back to the structure of a token, and specifically about the signature. Go ahead and play with the token again **here (https://jwt.io/#debugger)** . Change the signature. Notice that if the signature changes, that part of the token also changes according to the result of the algorithm. The signature is the key that is used to encode and decode the messages we want to keep hidden.

bcryptjs

Now we need to think about running some kind of algorithm to encrypt our password. One method is to use an npm package called `bcryptjs`. Bcrypt takes a value and applies an algorithm called a `salt` to it, returning a "hash value", or `hash`. The `hash` can then be decoded using the same algorithm. Essentially, the pattern is as follows, noting that we have kept a very basic explanation of how this process works:

1. The client sends a value to the server in its original form: the plain text password.
2. The server takes that value and applies the `salt` to the value a specified number of times.
3. Once this process is complete, the server passes the new `hash` value to the database to be stored.

   In the next module, we'll add it to our database and start hashing some passwords!

bcrypt Setup

02 - bcrypt setup

In this module we'll set up `bcrypt.js` in our application.

Import

We already have the `bcryptjs` package in our `package.json` file. Once installed, we have to add it to the database. Just like with `jwt`, create a new variable inside of `usercontroller.js`. For its declaration, use the `require` statement for `bcryptjs`.

```
var express = require('express');
var router = require('express').Router();
var sequelize = require('../db.js');
var User = sequelize.import('../models/user');
var bcrypt = require('bcryptjs'); //<---- ADD THIS
var jwt = require('jsonwebtoken');
```

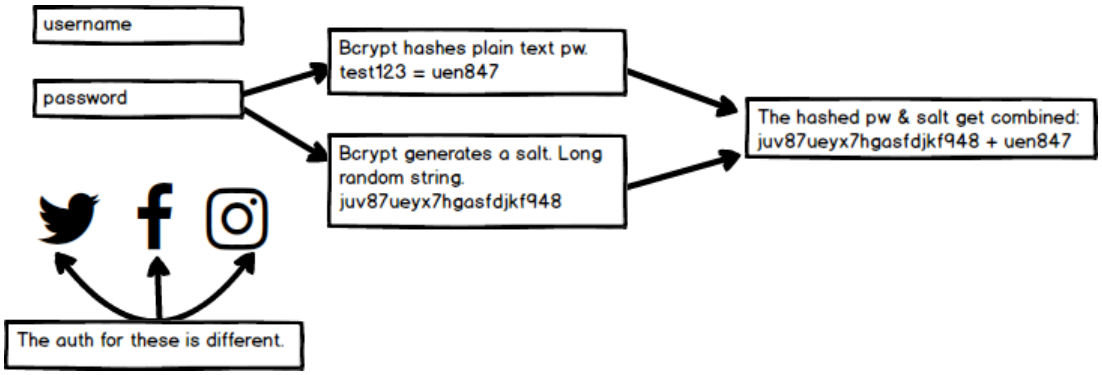
Adding bcrypt

Let's add bcrypt into the create method with the `pass` value for the `passwordhash` property:

```
router.post('/', function(req, res) {
        var username = req.body.user.username;
        var pass = req.body.user.password;

        User.create({
            username: username,
            passwordhash: bcrypt.hashSync(pass, 10)  //1 ADD THIS TO THE PROPERTY VALUE
        }).then(
            function createSuccess(user){
                var token = jwt.sign({id: user.id}, process.env.JWT_SECRET, {expiresIn: 60*60*24});

                res.json({
                        user: user,
                        message: 'created',
                        sessionToken: token
                });
            },
            function createError(err){
                res.send(500, err.message);
            }
        );
    });

module.exports = router;
```

Examining bcrypt.hashSync()

Here's a short diagram to give you an overview:

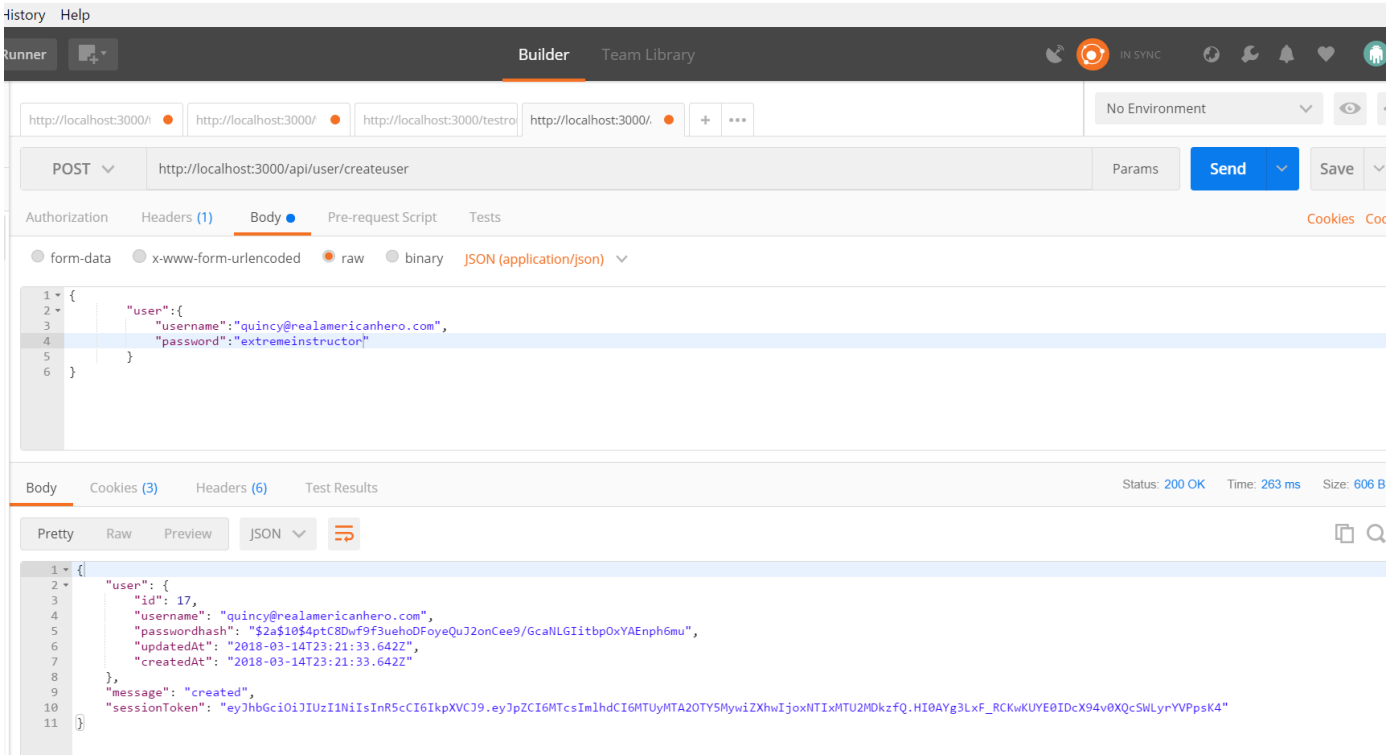


We're adding the `hashSync()` function to our new User object so that we don't store the password in a format that is easy to read. In our code, we supply the original password and tell bcrypt to use the salt 10 times.

Like JWT, there is a lot more to know and do with bcrypt. We are giving you enough to get started, but it would be wise to take a look through the **bcryptjs Docs** **(https://github.com/dcodeIO/bcrypt.js)**.

Postman and Postgres

Test in Postman:



Test in Postgres:

	16	passtest...	$2a$10$BsPYkGfzKcP6UescxWKXgecgjYCHw27pCU4C3PykAVooxoLks/Vk.	2018-03-...	2018-03-14 19:20:11.233-04
	17	quincy@r...	$2a$10$4ptC8Dwf9f3uehoDFoyeQuJ2onCee9/GcaNLGIitbpOxYAEnph6mu	2018-03-...	2018-03-14 19:21:33.642-04

Conclusion

At the moment, we have the ability to add new users to our database. Unfortunately, those users currently do not have a way back in after the initial signup, which defeats the purpose of having an account. In the next chapter, we'll setup a login route for existing users to access the database with their credentials.

Session Intro

00 - Session Intro

In this module, we'll discuss the concept of a session.

Scenario

Picture the following situation: You just signed up for a new email account. You've already given the new account to everyone you know and have deleted the old one. When you try to login, however, you discover that there's no ability to access the account after it's created, so you can't access your new email!
Here's the point: It's not worth signing up for something if you can't get back into it later. As mentioned previously, the only way to get into the database right now is to sign up a new user. Let's fix that with the ability to create new sessions for our users.

What is a Session?

When we talk about a session, we're talking about the length of time in which a user is doing something. Whether you're playing a video game, going for a job, or sitting down to work, each one of these activities can be called a session. For our purposes, a session is the length of time that you are allowed to access the database. In other words, the length of time a JWT token is valid. Remember that our token expires after 24 hours, so that is the maximum length of time that we can have a session last.

Establishing a Session

In order to establish a session, a user must receive a token. When a new user is added, we create that initial session by automatically giving them a token. To let them back into the database later, we have to verify that they are already in the `user` table and that their credentials match what we have stored. Only after this occurs will a token be issued, allowing them access. This token will either be revoked upon logging out or it will expire after 24 hours, at which point a new session must be established.

Sign In Method

01 - Sign In Method

In this module, we'll start making the sign-in method (signin) to create a new session.

The Code

Let's create our signin route. Add the following to the bottom of `usercontroller.js` file in `controllers`. Like the createuser route, each step will keep building on top of the previous until we have the finished product at the end:

```
        //7
router.post('/signin', function(req, res) {
            //1     //2       //3                                  //4
    User.findOne( { where: { username: req.body.user.username } } ).then(

        //5
        function(user) {
            if (user) {
                res.json(user);
            } else {
                res.status(500).send({ error: "you failed, yo" }); //6
            }
        }
    );
});
```

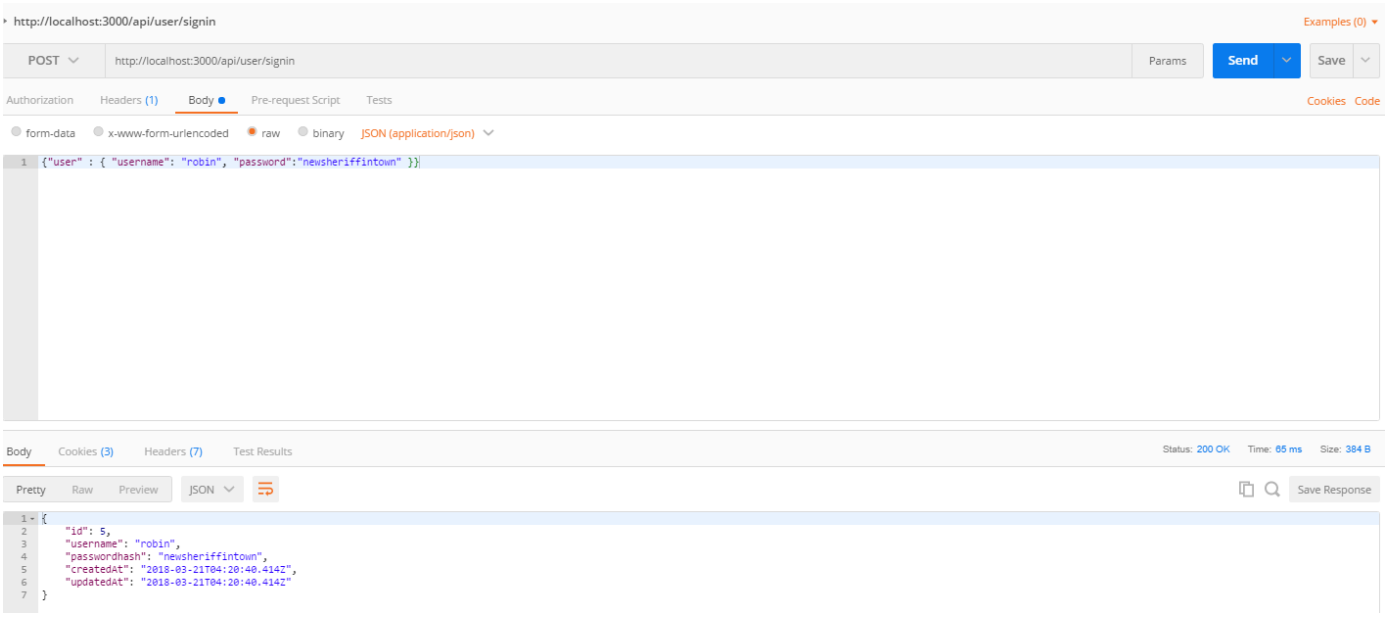
Analysis

Try reading the code above to see if you can get an intuitive sense of what is happening. Then, take a look at the following points for guidance or clarification:

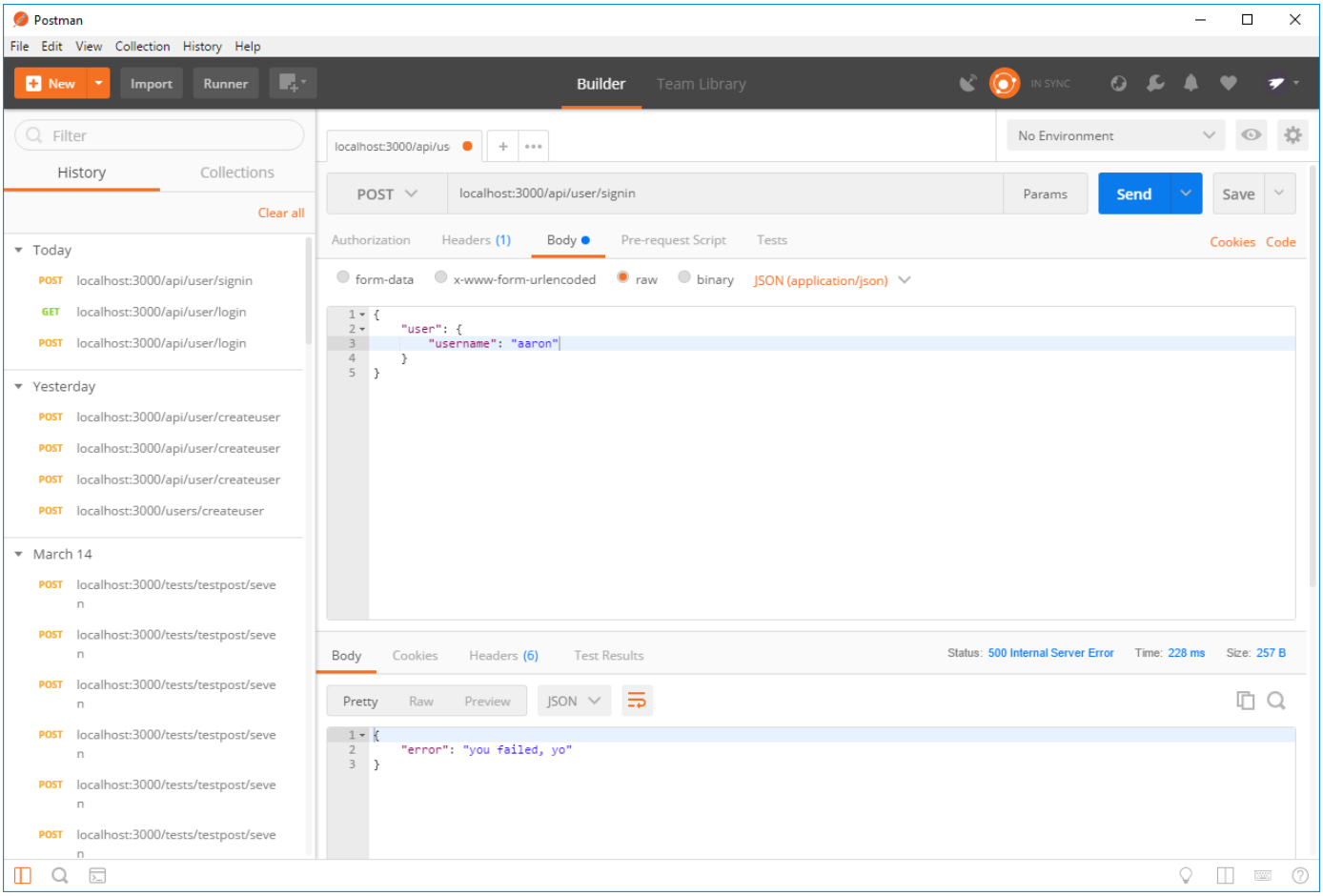
1. The `findOne()` method is a Sequelize method that does exactly what it says: it tries to find something within the database that we tell it to look for. This is called Data Retrieval. Check out the Sequelize docs **here (http://docs.sequelizejs.com/manual/tutorial/models-usage.html)** .
2. `where` is an object within Sequelize that tells the database to look for something matching its properties.
3. We're looking in the `username` column in the `user` table for one thing that matches the value passed from the client.
4. The promise is handled within the `.then()` function.
5. Here we have a function that is called when the promise is resolved, and if successful, sends the `user` object back in the response.
6. Function called if the promise is rejected. We print the error to the console.
7. We're sending data this time, so we use `router.post` instead of `router.get`.

Test

Start your server and open Postman. Send a request to the `signin` route with the username of one of the users in your database. NOTE: THIS USER SHOULD EXIST IN YOUR DB. You should see the user object print to the console:



Send another request, this time with a user not in the database. You should be hitting the error function and seeing something like this:



Sign In bcrypt

02 - Sign In Bcrypt

ADDING BCRYPT TO SIGNIN

In this module, we'll add bcrypt to our sign in request for security when creating a new session.

bcrypt.compare()

At the moment, we're only checking that the username matches something in the database. This would be a giant security issue, since the password doesn't even get checked! We are going to expand our `/signin` method here, so be aware that we are adding to the previous method where you see comments:

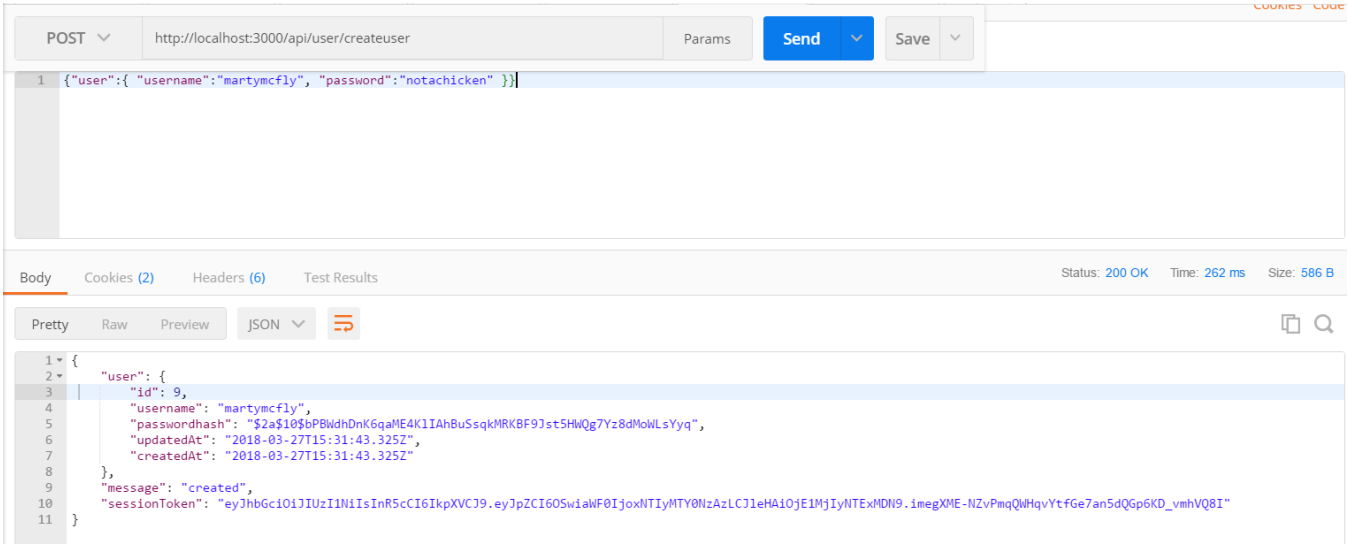
```
router.post('/signin', function (req, res) {
    User.findOne({ where: { username: req.body.user.username } }).then(
        function (user) {
            //1
            if (user) {
                      //2               //3                       //4           //5
                bcrypt.compare(req.body.user.password, user.passwordhash, function (err, matches) {
                    console.log("The value matches:", matches); //6
                });
            } else { //7
                res.status(500).send({ error: "failed to authenticate" });
            }
        },
        function (err) {
            res.status(501).send({ error: "you failed, yo" });
        }
    );
});
```

What Did We Just Do?

1. First we check to make sure that a match for the username was found.
2. Before, we used `bcrypt` to encrypt the password. Now, we use it to decrypt the hash value and compare it to the supplied password. This is a complex task, and we let the highly reputable and revered bcrypt package handle the algorithm for doing that. As a best practice, you shouldn't try to write this or use something that you have written. First of all, it will take months of your life to rebuild something that is already working. You can read more about `bcrypt.compare()` at the **npm registry (https://www.npmjs.com/package/bcryptjs)** .
3. Here we pull in the password value from the current request when the user is signing up.
4. This pulls the hashed password value from the database.
5. Run a callback function that will run on either success or failure of `compare`.
6. If the hashed password in the database matches the one that has been entered, print to the console that the password values match. Note that the `matches` variable is a boolean.
7. Handle situations where the match fails.

Test

1. Fire up Postman.
2. For the sake of clarity, let's be sure we can still create a new user:



3. Change the request endpoint to `/signin` instead of `/createuser` and press send, go back to the console since the output is a `console.log()` statement:

```
Executing (default): INSERT INTO "users" ("id","username","passwordhash","createdAt","updatedAt") VALUES (DEFAULT,'w4g4asdfw','$2a$10$wpv72xhVG/JsM1mKUvvBTe3Ek8SrCSSA2epBYO.ZeaC
oDA.egeC/u','2018-03-29 19:23:38.934 +00:00','2018-03-29 19:23:38.934 +00:00') RETURNING *;
Executing (default): SELECT "id", "username", "passwordhash", "createdAt", "updatedAt" FROM "users" AS "user" WHERE "user"."username" = 'w4g4asdfw' LIMIT 1;
The value matches: true
```

4. We aren't able to view the response in Postman because we are missing something! You'll find out what it is in the next section.

Sign In JWT

03 - Sign In JWT

In this module, we will write the code that will return a token in response to our sign in request.

The Code

As of right now, we are not receiving a token back from our sign in request. We need this to establish the length of our current session. The code we will add will replace the `console.log()` inside of the `compare` method:

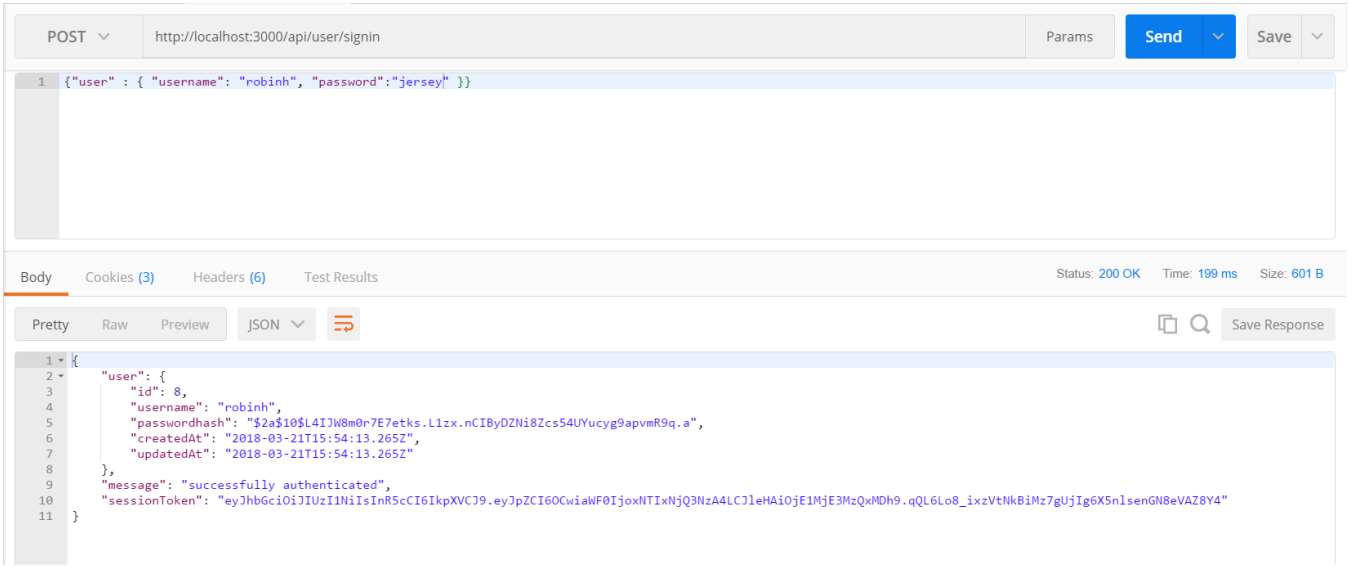
```
router.post('/signin', function(req, res) {
    User.findOne( { where: { username: req.body.user.username } } ).then(
        function(user) {
            if (user) {
                bcrypt.compare(req.body.user.password, user.passwordhash, function(err, matches){
                    //1
                    if (matches) {
                        //2
                        var token = jwt.sign({id: user.id}, process.env.JWT_SECRET, {expiresIn: 60*60*24 });
                        res.json({  //3
                            user: user,
                            message: "successfully authenticated",
                            sessionToken: token
                        });
                    }else { //4
                        res.status(502).send({ error: "you failed, yo" });
                    }
                });
            } else {
                res.status(500).send({ error: "failed to authenticate" });
            }
        },
        function(err) {
            res.status(501).send({ error: "you failed, yo" });
        }
    );
});
```

Analysis

1. Here we use the callback function from the `compare()` method. If the username and password are a match, this will be set to true, and the expression in the conditional will execute.
2. Upon success, we will create a new token for the session. Note that this code uses the same `jwt.sign` method that we used upon sign up. We will let you review that code if you need clarification.
3. We return the user object with a success message and sessionToken.
4. If the passwords don't match or the username is not correct, we send a response telling the client that authentication did not occur.

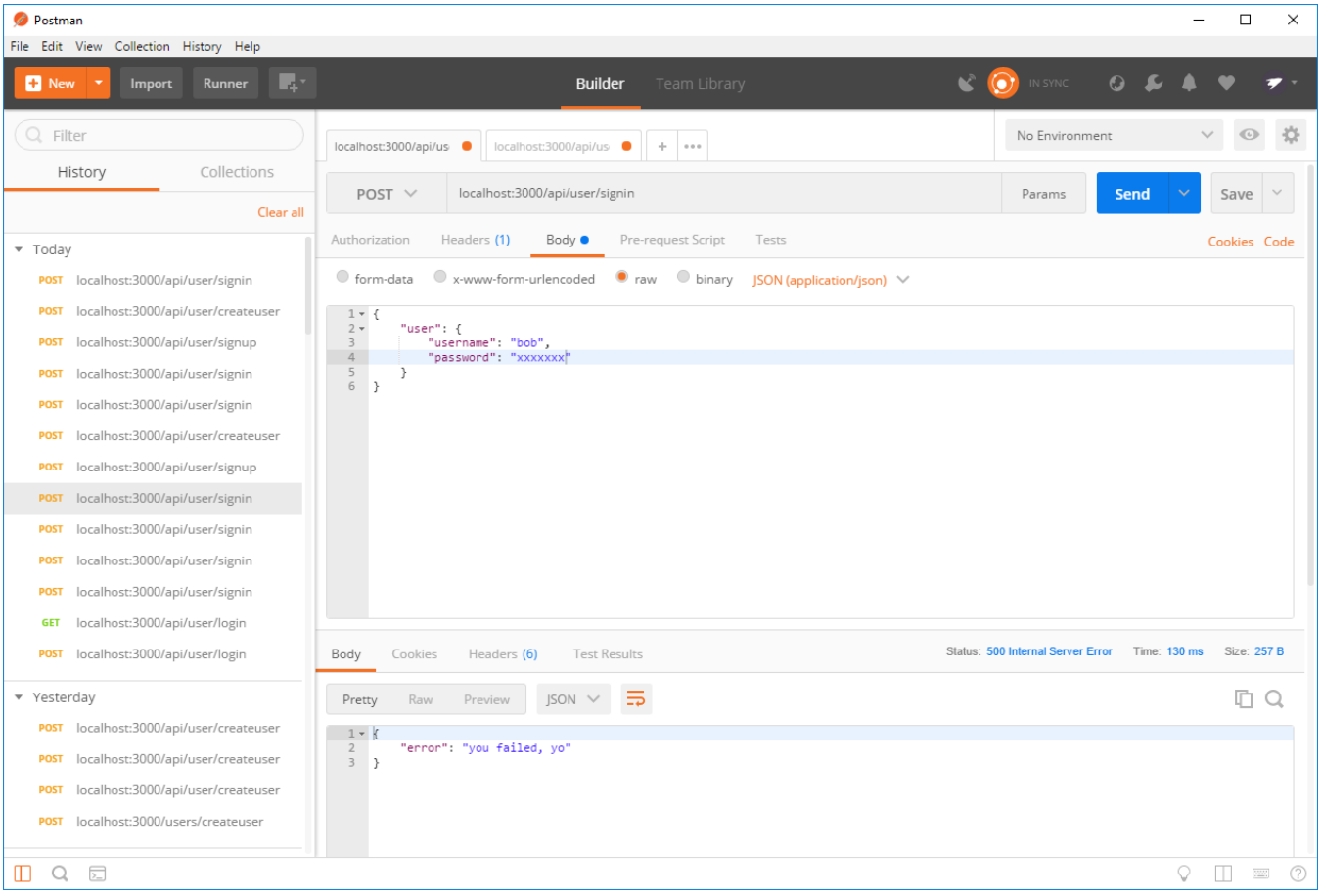
Test

1. Run a few requests in Postman.
2. Create a new user with a DIFFERENT username than you have previously used. NOTE: findOne will only return the FIRST instance of the username that it finds in the DB, so it's important that you create a new unique username.
3. Log in with the username you just created and the corresponding password:



4. Stop and notice the token in the response.

5. Try logging in with a user that doesn't exist or with a user exists with the wrong password:



Quick Summary

1. Here we've added bcrypt to compare passwords entered by the client with the password in the database.
2. If the sign in is successful, we send a token back to the user.
3. If the sign in is not successful, we send an authentication error.

Test Client HTML

01 - Test Client HTML

In this module, we're going to set up the front-end client HTML for testing our requests in the DOM.

Overview

In this module, we'll add some basic HTML and JavaScript on the front-end to allow us to learn to interact with our server in a practical way. It's important to know that Postman is acting as a proxy-client for us, a stand in until we get something built on the front-end. As we transition to working with a custom client, there are some additions that need to be made to our server that Postman didn't need, including more middleware and headers. We'll show you those things in future modules.

File Structure

Please add a client folder underneath the server folder:

```
    └── 5-Express Server
            └── server
            └── client
                └── 01-scripts.js
                └── index.html
```

Initialize NPM

In order for our client to run on `localhost:8080`, we'll need to add `http-server`. Let's set up npm to allow this to happen:

1. cd into the `client` directory.
2. Run `npm init`. Make sure a `package.json` file is created.
3. Run `npm install http-server --save`.
4. Your folder structure should now look like this:

```
    └── Node-Server
            └── server
            └── client
                └── 01-scripts.js
                └── node_modules
                └── index.html
                └── package-lock.json(this might be created. might not though)
                └── package.json
```

index.html

Copy the following code into the `index.html` file. Before we talk more about it, use what you know from previous experience with client-side programming to see if you can guess what this code is doing:

```
<!DOCTYPE html>
<html>

<head>
    <title>Fullstack Starter</title>
    <!--1-->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" integrity="sha384-Gn5384xqQ
1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
        crossorigin="anonymous">
</head>

<body>
    <div class="container">
        <h1>Full Stack Starter</h1>
```

```
        <p>This app is meant as a guide for learning how to use HTTP with Fetch calls to an Express API using Sequelize and Postgre
s.
            Some methods print to the console and some produce material in the DOM. Either way, these chunks of code are
            meant to be a guide when working with the DOM and data for a homespun Express server.</p>

        <!--2-->
        <table class="table table-striped">
            <thead>
                <tr>
                    <th scope="col">Step</th>
                    <th scope="col">Endpoint</th>
                    <th scope="col">Action</th>
                    <th scope="col">Result</th>
                </tr>
            </thead>
            <tbody>
                <tr>
                    <th scope="row">1</th>
                    <td>GET
                        <code>/test/helloclient</code>
                    </td>
                    <td>
                        <button onclick="fetchHelloDataFromAPI();">Hello Client</button>
                    </td>
                    <td>See Console</td>
                </tr>
                <tr>
                    <th scope="row">2</th>
                    <td>POST
                        <code>/test/one</code>
                    </td>
                    <td>
                        <button onclick="postToOne();">POST to /one</button>
                    </td>
                    <td>See Console</td>
                </tr>
                <tr>
                    <th scope="row">3</th>
                    <td>POST (arrow function)
                        <code>/test/one</code>
                    </td>
                    <td>
                        <button onclick="postToOneArrow();">POST to /one</button>
                    </td>
                    <td>See Console</td>
                </tr>
                <tr>
                    <th scope="row">4</th>
                    <td>POST Data
                        <code>/test/seven</code>
                    </td>
                    <td>
                        <button onclick="postData();">Post</button>
                    </td>
                    <td>
                        <ul>
                            <li>
                                <bold>Posted data:</bold>
                                <span id="test-data"></span>
                            </li>
                            <li>
                                <bold>At:</bold>
                                <span id="created-at"></span>
                                </p>
                            </li>
                        </ul>
                    </td>
                </tr>
                <tr>
                    <th scope="row">5</th>
                    <td>GET and display json
                        <code>/test/one</code>
```

```
                    </td>
                    <td>
                        <button onclick="fetchFromOneDisplayData();">Display data</button>
                    </td>
                    <td>
                        <ul id="getjson">
                        </ul>
                    </td>
                </tr>
            </tbody>
        </table>
    </div>

    <!--3-->
    <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN"
        crossorigin="anonymous"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js" integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q"
        crossorigin="anonymous"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js" integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmYl"
        crossorigin="anonymous"></script>


    <script src="01-scripts.js"></script>
</body>

</html>
```

Overview of the HTML

This tutorial assumes you have a basic grasp on HTML, so we'll briefly summarize what we've done. Overall, we're create a single HTML file that will allow us to test our endpoints, but there are a few items to note here: 1. **Head:** We're not doing any styling, so there are no links to custom CSS files in the head. However, we are going to use Bootstrap, so we did add a CDN link to the Bootstrap CSS file that is needed for building tables. 2. **Body Table:** We're building a table that will show us how to to access and display our data from the server in the DOM. This just makes an easy-to-use button system for testing our endpoints. 3. We add the Bootstrap JS CDN and our own custom JavaScript file, which we'll use next.

Test

To test this, we'll work in a web browser using `http-server`. If you haven't installed this package globally, do the following first: 1. Navigate to your client folder in your terminal window. 2. Run the following command: `npm install http-server --save`. This will add `http-server` to your package.json so anyone cloning your project will install it. Additionally, if you install a package locally inside a project that you also have installed globally, the local version will be used unless you override it. This usually won't cause any issues, but it's good to be aware that this is happening. 3. In your `package.json`, add the following to the `scripts` property (or create it if it's not already there). `http-server` won't run without it:

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "http-server" //<---ADD THIS
  },
```

Once you have `http-server` setup either locally or globally, follow these steps to get the client running: 1. cd into your client directory. 2. Run `npm start`. 3. Open a browser and go to localhost:8080. 4. You should see the following:

Full Stack Starter

This app is meant as a guide for learning how to use HTTP with Fetch calls to an Express API using Sequelize and Postgres. Some methods print to the console and some produce material in the DOM. Either way, these chunks of code are meant to be a guide when working with the DOM and data for a homespun Express server.

Step	Endpoint	Action	Result
1	GET `/test/helloclient`	Hello Client	See Console
2	POST `/test/one`	POST to /one	See Console
3	POST (arrow function) `/test/one`	POST to /one	See Console
4	POST Data `/test/seven`	Post	• Posted data: • At:
5	GET and display json `/test/one`	Display data	

Test Client JS

02 - Test Client JS

In this module, we're going to start adding the JavaScript functions that will enable us to make client-side requests to our server.

Overview

So far, we've been using Postman to test all of our endpoints. Postman is great, but we want to see how to get the data and start showing it in the DOM. Again, we're just keeping the DOM stuff in a simple Bootstrap table for now, and we're going to be focused solely on getting data into the DOM. We can worry about prettying things up another day.

Server Addition

Just for reorientation, let's add something a little extra in our server:

1. Go to your `testcontroller.js` file in the `controllers` folder.
2. Add the following route:

```
/************************
* GET:  Get simple message from server
***********************/
router.get('/helloclient', function (req, res) {
res.send('This is a message from the server to the client.')
})
```

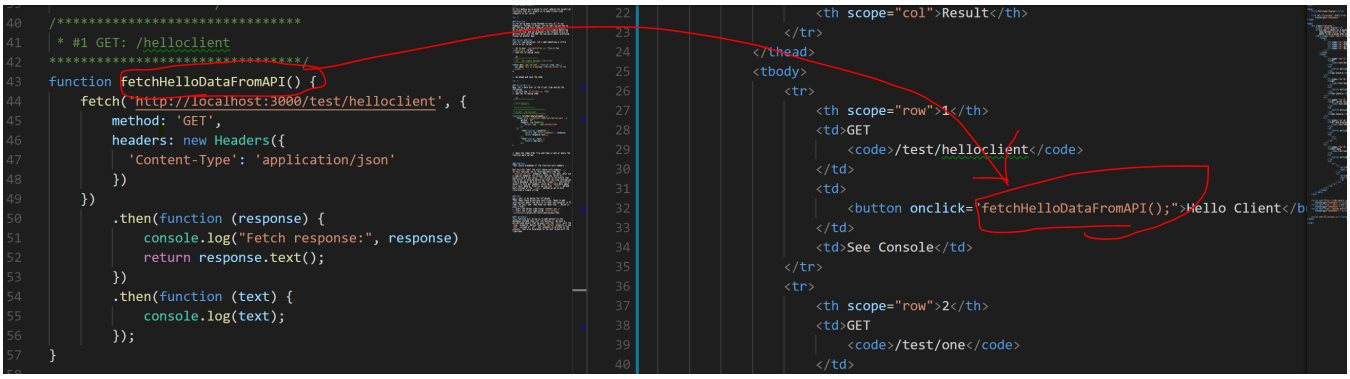
3. Go ahead and save the code.

01-scripts.js

Now, let's move over to the client side and do the following: 1. Go into the `01-scripts.js` file. 2. Add the following code:

```
function fetchHelloDataFromAPI() {
    fetch('http://localhost:3000/test/helloclient', { //1
        method: 'GET',
        headers: new Headers({ //2
          'Content-Type': 'application/json'
        })
    })
        .then(function (response) {
            console.log("Fetch response:", response)
            return response.text(); //3
        })
        .then(function (text) {
            console.log(text);
        });
}
```

1. Open the `index.html` file and take a look at where the function gets called:



```
/*******************************
 * #1 GET: /helloclient
 *******************************/
function fetchHelloDataFromAPI() {
    fetch('http://localhost:3000/test/helloclient', {
        method: 'GET',
        headers: new Headers({
          'Content-Type': 'application/json'
        })
    })
        .then(function (response) {
            console.log("Fetch response:", response)
            return response.text();
        })
        .then(function (text) {
            console.log(text);
        });
}
```

```
            <th scope="col">Result</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <th scope="row">1</th>
            <td>GET
                <code>/test/helloclient</code>
            </td>
            <td>
                <button onclick="fetchHelloDataFromAPI();">Hello Client</b
            </td>
            <td>See Console</td>
        </tr>
        <tr>
            <th scope="row">2</th>
            <td>GET
                <code>/test/one</code>
            </td>
```

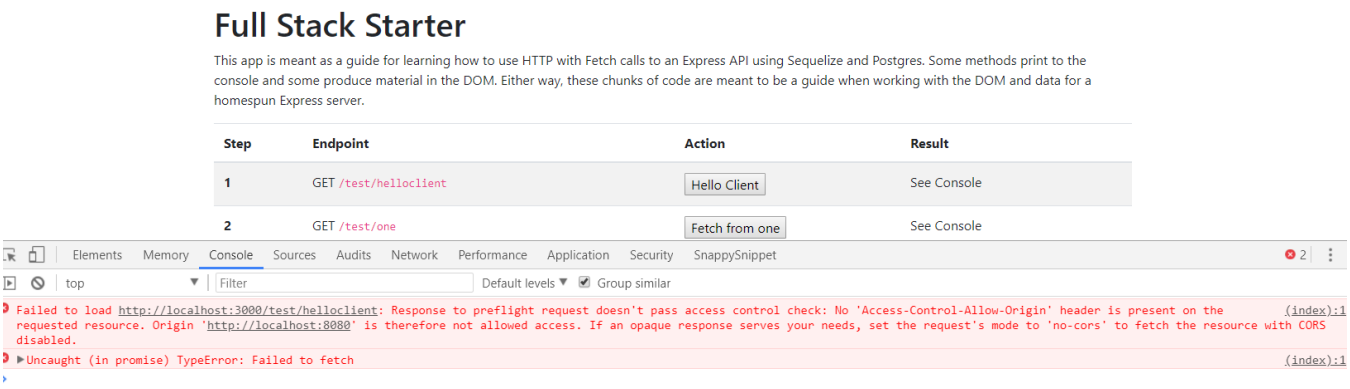
Analysis

1. Test endpoint with fixed value to verify server works.
2. Send our headers to the server with the `Headers()` constructor object. We'll talk more about this in a later module.
3. The value received is a string, not a JSON object, so `.text()` is used instead of `.json()`

Test

Let's test it by doing the following:

1. In your console window in VS Code, click the `+` sign to open a new console window. You can switch between windows using the dropdown next to the `+`. You can also right-click a folder in the File Explorer and select `Open in Command Prompt`(Windows) or `Open in Terminal` (Mac) to open a console window in that specific folder.
2. Navigate to your server folder and start the server side using `nodemon app.js`.
3. *NOTE: If you client is already running, skip this step.* Open another new terminal window, navigate to the client folder, and start the client side using `http-server -o`. This will start the client and automatically open a new tab in your browser at 127.0.0.1:8080 (same as localhost:8080).
4. Click any of the buttons on screen. You should see an error:



CORS

When a client and server are both running at the same time, they run on different ports. By default, a server won't recognize any transmission coming from a different URL than its own, and it will refuse to acknowledge the request. This is known as a `CORS` error, which stands for **Cross Origin Resource Sharing**. We can fix errors like this with files known as *middleware*, which we'll talk more about in the next chapter.

Middleware Intro

03 - Middleware intro

In this module, we'll start to address the error that we ran into in the last module with our client request.

MIDDLEWARE

Sometimes our client and server don't communicate well. Data can get lost or damaged during transmission, unexpected extra data could be included, or a host of other issues could occur. This module will introduce you to a type of code that is designed to help these transmissions go as smoothy as possible: Middleware.

Middleware lives in the space between the client and the server. Its job is to facilitate the process of sending data between the two, as well as ensuring that no unwanted data accidentally gets passed along with the request or response. It can also be used to help prevent external users from accessing transmissions that could then be used to compromise our system. Middleware is an additional checkpoint that must be successfully navigated in order for our transmission to reach its desired destination.

In the previous module, we briefly touched on the idea of `CORS`. One of the main purposes of middleware is to tell the server to allow transmissions coming from outside locations. We can choose to allow any outside traffic, no outside traffic, or specific locations. We do this with special properties called `headers`, which we'll discuss in the next module.

Headers Intro

04 - Headers intro

In this module, we'll use Express to begin allowing CORS for client and server request/response cycle.

Headers

Headers are sent by the client along with the request. They contain special instructions for the server. For more information on headers and CORS, take a look at the **Mozilla docs** (https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS). A word of caution: there is a lot of information packed into this page, and much of it focuses on things that we haven't covered yet. Take it slow and do your own research, too.

1. Go into the server folder, create a new folder called `middleware`, and add the following files:

```
└── 5-Express Server
        └── server
            └── controllers
            └── middleware
                └── headers.js
            └── models
        └── client
```

2. Go into `headers.js` and add the following code:

```
//1                       //2
module.exports = function(req, res, next){
 //3                    //4
 res.header('access-control-allow-origin', '*');
 res.header('access-control-allow-methods', 'GET, POST, PUT, DELETE'); //5
 res.header('access-control-allow-headers', 'Origin, X-Requested-With, Content-Type, Accept, Authorization'); //6
 //7
 next();
};
```

Analysis

1. `module.exports` allows us to export this module to be used in another file.
2. `req` refers to the request from the client, specifically focusing on any headers present on the request object. `res` refers to the response and will be used to present which types of headers are allowed by the server. `next` will be covered more in a moment.
3. We call `res.header` so that the server will respond with what kind of headers are allowed in the request.
4. We use the specific `access-control-allow-origin` header to tell the server the specific origin locations that are allowed to communicate with the server. The `*` is known as a `wild-card`. It means that everything is allowed. In this setting, it's saying that requests originating from any location are allowed to communicate with the database.
5. These are the HTTP methods that the sever will allow to be used. Postman allows you to send 15 different HTTP requests; our server will only accept these four.
6. These are specific header types that the server will accept from the client. Remember from our earlier testing we sent a `Content-Type` header to the server. Without this header, our request would not have worked. You can find more information on this and other headers on MDN, and we will talk about them more in the future as well.
7. `next` sends the request along to its next destination. This could be the API endpoint or another middleware function designed to do something else. Let's talk a little bit more about `next`.

next()

`next()` tells the middleware to continue its process. With the above example, `next()` takes the request object and passes it on the endpoint on the server. Not including the `next()` would cause the application to break, as the server doesn't know what to do after sending the header. We could also use `next()` to provide additional headers if we want further restrictions on our server.

Server Update

05 - Server Update

In this module, we'll add header files in our server.

Overview

We added some routes to `app.js` earlier but didn't cover what was really happening there. In this module, we'll add to the `app.js` file, giving it the ability to use middleware. We'll also try testing out the additional routes we added earlier, as well as use the test client to see how the client and server sides work together.

Referencing the Middleware

We exported the middleware, so now we need to use it when we spin up our server. Follow the steps: 1. Go into `app.js`. 2. Add the following line of code under the `bodyParser` variable:

```
sequelize.sync(); // tip: {force: true} for resetting tables
app.use(bodyParser.json());
app.use(require('./middleware/headers')); //1 Add it here.
app.use('/test', test);
app.use('/api/user', user);
app.listen(3000, function(){
    console.log('App is listening on 3000.')
});
```

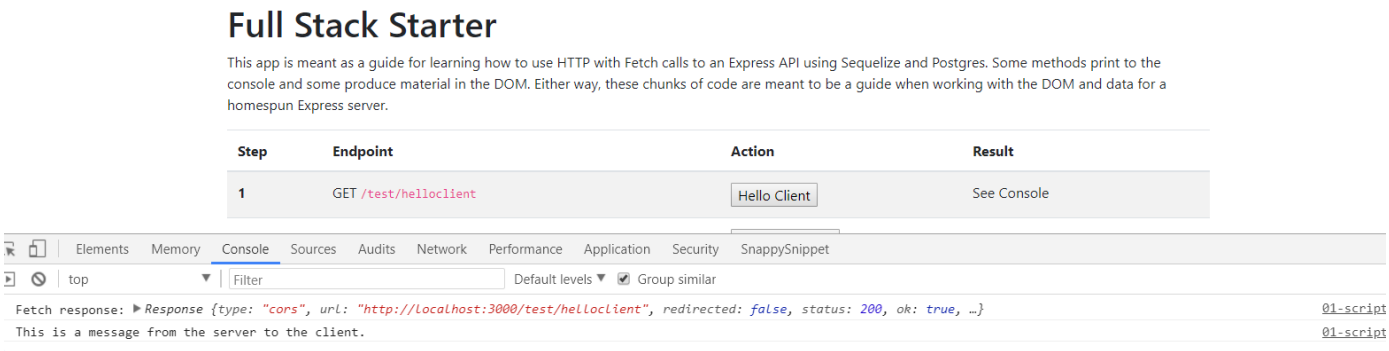
Analysis

1. Here we activate our headers in the `app.js`. Keep in mind that this in order, so the file will be read sequentially, which means that the headers must come before the routes are declared.

Test Client

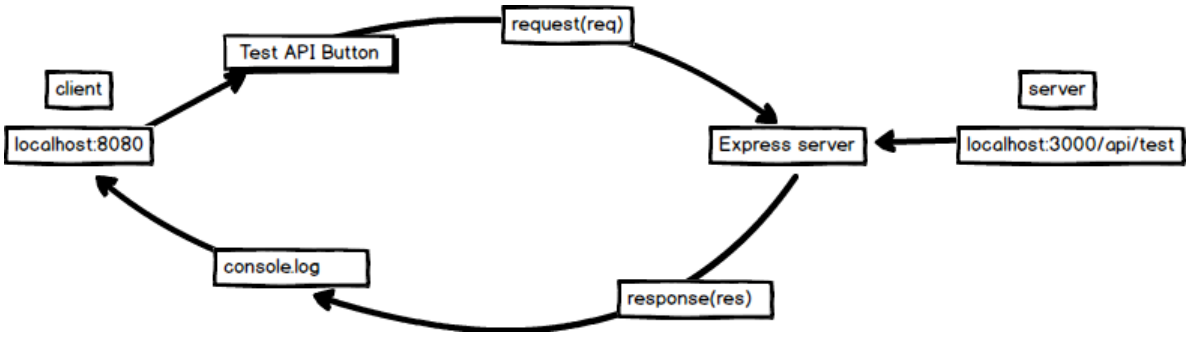
Let's test this in our client now with the following steps:

1. Your server should still be running, so we just need to start the client. Switch to the client terminal window and start it.
2. Go to your client and try clicking on the button that says **Hello Client**. You should see the result print to the console. Please note this is the only button that currently works.



Further Study

Look at this picture of the client-server relationship:



Explain the entire process, from the client creating of the request to receiving the response back from the server.

Test Post

06 - Test Post

In this module, we'll write another client side function that access our server.

Overview

This time, we'll be testing our POST request from the `test/one` endpoint that we already built, which will pull from the database instead of a pre-defined value. We'll chain promises together to create a flow chart for our `fetch`.

Scripts

Let's add the function now. It should go directly below the first function in `01-scripts.js`:

```
/*****************************
 * 2 POST long hand: /one
 *****************************/
function postToOne(){
    var url = 'http://localhost:3000/test/one';

    fetch(url, {
      method: 'POST',             //1
      headers: new Headers({
        'Content-Type': 'application/json'
      })
    }).then(
        function(response){   //2
            return response.text()
        })
    .catch(
        function(error){   //3
            console.error('Error:', error)
        })
    .then(
        function(response){   //4
            console.log('Success:', response);
        })
}
```

Quick Summary

We are not seeing new stuff here, but let's do a quick summary: 1. We are fetching the url. The route in the server handles a `POST` request, so our method type is `POST`. Remember that these two must match. If a route takes a `POST` request, then the declared method in the request should `POST`. 2. We pass the response into a Promise that returns the response as plain text. (We'll use more JSON later) 3. We handle an error, if an error comes back. 4. In the final `then()`, we simply print the plain text response to the console. This section is where we can do some DOM set up.

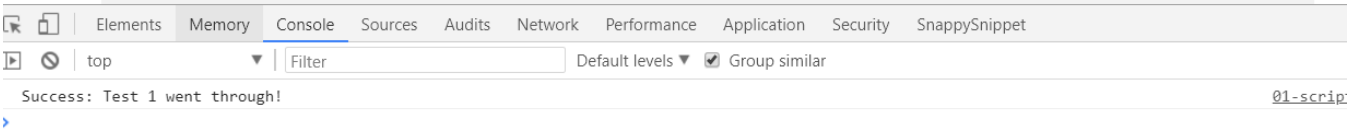
Test

1. Make sure that both your client and server are running.
2. Go to `localhost:8080`
3. Click the `POST to /one` button.
4. You should see the following success message:

Full Stack Starter

This app is meant as a guide for learning how to use HTTP with Fetch calls to an Express API using Sequelize and Postgres. Some methods print to the console and some produce material in the DOM. Either way, these chunks of code are meant to be a guide when working with the DOM and data for a homespun Express server.

Step	Endpoint	Action	Result
1	GET `/test/helloclient`	Hello Client	See Console
2	POST `/test/one`	POST to /one	See Console



Test Post Refactor

07 - Test Post Refactor

In this module, we'll refactor our last method so that it is more streamlined and using ES6 principles.

Overview

Right below the `postToOne` method, let's write another method. We'll test that same endpoint, except we'll write our function in a more concise form. This is helpful to know because it is common to see the use of arrow functions as we look ahead to React. Also, it can make our code much more streamlined.

Scripts

Let's add the function now. It should go directly below the first function in `01-scripts.js`:

```
  /*************************************
 * 3 POST /one : Arrow Function
*************************************/
function postToOneArrow(){
    var url = 'http://localhost:3000/test/one';

    fetch(url, {  //1
      method: 'POST',
      headers: new Headers({
        'Content-Type': 'application/json'
      })
    }).then(res => res.text()) //2
    .catch(error => console.error('Error:', error)) //3
    .then(response => console.log('Success:', response)); //4
}
```

Quick Summary

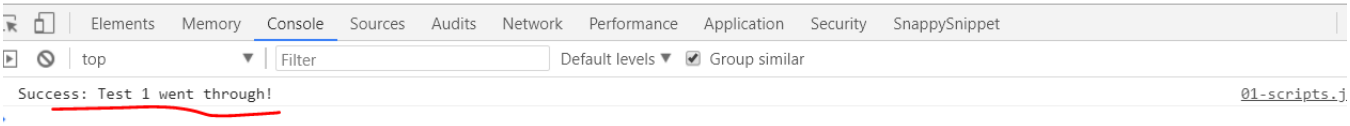
This is the same function as the one above it that we wrote in the last module, only we use arrow functions instead of callbacks. You can see how much more simplified and easier to read this can be, saving us from potential "Callback Hell". Our function is doing the same thing:

1. We're reaching out to an endpoint with a POST request. We add the appropriate headers.
2. We are asking for a plain text response.
3. We handle an error, if there is one.
4. In the end, we simply print the data to the console.

Test

1. Make sure that both your client and server are running.
2. Go to `localhost:8080`
3. Click the `POST to /one` button in Step #3.
4. You should see the following success message:

Step	Endpoint	Action	Result
1	GET `/test/helloclient`	Hello Client	See Console
2	POST `/test/one`	POST to /one	See Console
3	POST (arrow function) `/test/one`	POST to /one	See Console
4	GET and display json `/test/one`	Display data	



Post Data

08 - Post Data

In this module, we'll send some data to our test route to store in the database. Then we'll show some data in the DOM.

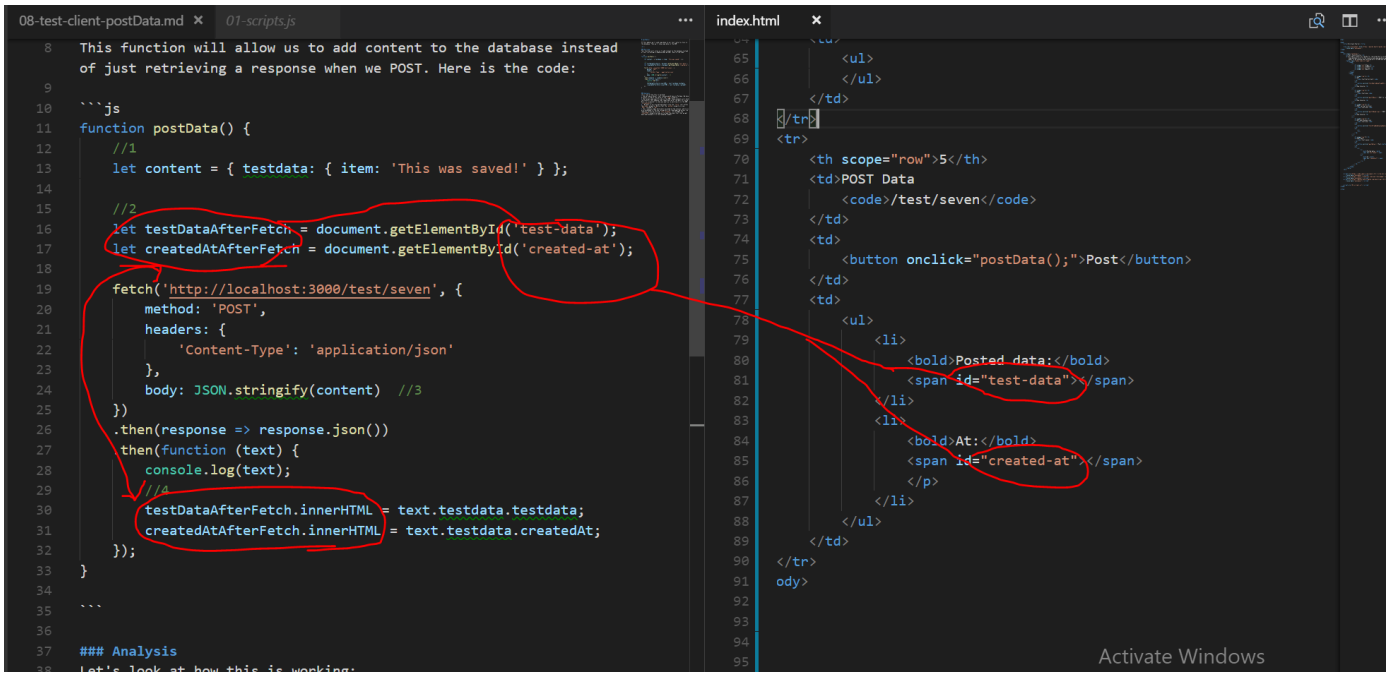
Overview

This function will allow us to add content to the database instead of just retrieving a response when we `POST`. Here is the code that should go after `postToOneArrow()` in `01-scripts.js`:

```
function postData() {
    //1
    let content = { testdata: { item: 'This was saved!' } };

    //2
    let testDataAfterFetch = document.getElementById('test-data');
    let createdAtAfterFetch = document.getElementById('created-at');

    fetch('http://localhost:3000/test/seven', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(content)  //3
    })
    .then(response => response.json())
    .then(function (text) {
        console.log(text);
        //4
        testDataAfterFetch.innerHTML = text.testdata.testdata;
        createdAtAfterFetch.innerHTML = text.testdata.createdAt;
    });
}
```

Analysis

Let's look at how this is working: 1. We set up an object, just like we would have in Postman. We have a preset string as the value of the `item` property. 2. We target some specific `ids` in the DOM. These elements will hold the value of the response that comes back after the post is stored. Here's a screenshot to show you what we're pointing to in the DOM.

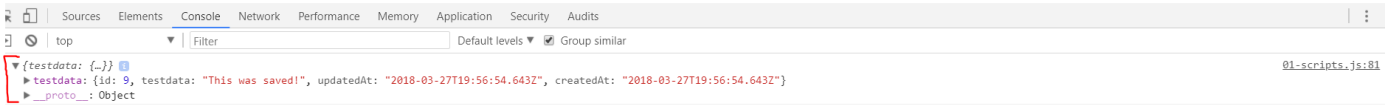


1. We pass in our pre-defined object into the `fetch` call within the `body` property. Notice that the `method` property is now `POST` instead of `GET`. 4. Our response comes back and is printed to the console, and it is also displayed to the user along with the timestamp. We access the separate values by calling `text.testdata`. In the DOM, the `innerHTML` property allows us to take the plain text that we get back and display it in the targeted element.

Test

Let's test the code to ensure that things are working: 1. Make sure that both your client and server are running. 2. Go to localhost:8080. 3. Press the POST button. Open the console. 4. This is how it should look:

Step	Endpoint	Action	Result
1	GET /test/helloclient	Hello Client	See Console
2	POST /test/one	POST to /one	See Console
3	POST (arrow function) /test/one	POST to /one	See Console
4	POST Data /test/seven	Post	• Posted data: This was saved! • At: 2018-03-27T19:56:54.643Z
5	GET and display json /test/one	Display data	



Fetch From One

09 - Fetch From One

In this module, we'll create and access the `/one` endpoint and do a `GET` request to access data for display in the DOM.

Server Addition

When writing applications, it's common to have to toggle between the front-end and back-end. With that being said, let's add another route in the `testcontroller.js` file. This can be anywhere in the file, but we'll put it just before `module.exports = router`.

```
/***********************
 * GET:  /one
 ***********************/
router.get('/one', function(req, res) {

  TestModel
    .findAll({ //1
        attributes: ['id', 'testdata']
    })
    .then(
        function findAllSuccess(data) {
            console.log("Controller data:", data);
            res.json(data);
        },
        function findAllError(err) {
            res.send(500, err.message);
        }
    );
});
```

Analysis

1. Notice that we find the `attributes` for two of the columns: `id` & `testdata`. This is part of sequelize. If you are querying an entire table, you can choose which columns you want to grab from. The other columns will not be queried, which can save time for a giant table.

Be sure to save the changes to your server.

Overview

Let's go back to the client and into the `01-scripts.js` file. Right below the `postData` method, let's write another method. This time we'll pull in the data and think through the starter logic of showing it in the DOM. Here is the code to be added:

```
/*************************************
 * 4 GET FROM /ONE - Display Data
**********************************/
function fetchFromOneDisplayData(){
    //1
    let url = 'http://localhost:3000/test/one';
    let dataView = document.getElementById('display-one');

    //2
    fetch(url, {
      method: 'GET',
      headers: new Headers({
        'Content-Type': 'application/json'
      })
    }).then(
        function(response){
            return response.json()
        })
    .catch(
```

```
        function(error){
            console.error('Error:', error)
        })
    .then(
        function(results){
            let myList = document.querySelector('#getjson'); //3

            for (r of results){  //4
                console.log('Response:', r.testdata); //5
                var listItem = document.createElement('li');  //6
                listItem.innerHTML = r.testdata; //7
                myList.appendChild(listItem); //8
            }
        })
}
```

Summary

Let's break this down:

1. We set up our URL in one variable and target the `data-one` id in the DOM in another one.
2. We create a `fetch()` with `Headers` and therequest method of `GET`. There are also chained promises that handle the data when it returns or handle an error if one comes back.
3. Inside the final `.then()`, we are going to work towards showing the returned data in the DOM. We start by targeting the `getjson` id in the DOM and attaching the value to the `myList` variable.
4. We set up a `for of` loop.
5. We write a `console.log()` statement to show how we can access the values that come back in the object inside the response.
6. We create another variable called `listItem`. The `createElement()` method will create that type of element in the DOM. In this case, we create a list item, `li`, every time we iterate.
7. Each time we iterate, we store the value of `r.testdata` in the newly create `li`.
8. We call `appendChild` on `myList`, which means that each time we iterate we put the `li` into the unordered list.

Quick Summary for the DOM Work

Just to succinctly summarize the last `then()`:

- We target a list, `myList`.
- We iterate over the response object with a `for of` loop.
- Each time we iterate, we create a list item.
- The value gets stored in the `innerHTML` of the `li`.
- We append the list item to an unordered list.
- We continue until we get to the end of the response object.

Test

1. Make sure that both your client and server are running.
2. Go to `localhost:8080`
3. Click the `Display Data` button in Step #5.
4. You should see the following success message:

1	GET `/test/helloclient`	Hello Client	See Console
2	POST `/test/one`	POST to /one	See Console
3	POST (arrow function) `/test/one`	POST to /one	See Console
4	POST Data `/test/seven`	Post	• Posted data: This was saved! • At: 2018-03-28T12:55:02.186Z • Test data for endpoint two • step 3 is cool • step 4 • step 5 • step 6 • step 7 • This was saved!
5	GET and display json `/test/one`	Display data	

Intro to Authenticated Routes

01 - Intro to Authenticated Routes

In this module, we'll begin working with authenticated routes by creating routes that require a valid token to access.

Overview

With Postman, we have the ability to create an account, login to an account, and receive a token. In a little while, we will use those routes in the DOM. However, for now, let's do some server work and prepare it for authenticated requests.

What is an Authenticated Route?

An authenticated route is another way of saying a **protected** route. There are parts of our database and site that we only want certain people to be able to access, and only certain parts that those certain people are allowed into. When you login to your email or a site like Facebook, you're being taken to an authenticated route: your email inbox, your Facebook feed, etc. While their authentication processes are undoubtedly far more advanced and complicated than what we'll do, the idea is the same: keeping out any unauthorized traffic or malicious users.

How Do We Do It?

We need to write a middleware function that acts as a gate between the client and the server. This middleware is going to look for and look at our token in the request. If the request has a token, it's allowed to pass through the gate to reach the server and get data from Postgres to be returned or saved for a specific user. If not, the request is rejected.

Think back to the example about the bar from earlier. In this context, the bar is the route we want to authenticate. After you pay your cover and get your stamp, you go inside and order a drink from the bartender. First, the bartender has to check that you have a stamp, then check your ID to make sure that you're old enough to drink. This second step is the new middleware function that we're going to create: validation.

We can even take this one step further: Let's say the bar uses different stamps for different nights of the week. The doorman was only checking to make sure that you had a stamp; the bartender will check to make sure it's TONIGHT's stamp, not the previous night's. Remember that when we create our tokens, we give them an expiration date, so our new middleware function needs to not only verify that the token belongs to the user that it was assigned to, but also verify that the token is still valid.

Let's do a little bit of server setup now, and then make some changes to our client.

server/models/authtest.js

In order to try out using authenticated routes, we need a new table in our database that we can place behind an authentication barrier. Follow these steps:

1. In the `models` folder in the server, add an `authtest.js` file.
2. Add the following code inside of the file:

```
module.exports = function(sequelize, DataTypes) {
        return sequelize.define('authtestdata', {
            authtestdata: DataTypes.STRING,
            owner: DataTypes.INTEGER
        });
};
```

3. Notice that we will be providing two properties: `authtestdata` and `owner`.
4. Think of `authtestdata` as a string like `testData`.
5. The `owner` is a number, a foreign key, that will point to a specific `user` on the `users` table.

Validate Session

02 - Validate Session

In this module, we'll construct a file that will check to see if the request has a token attached.

Files

Please add the following to your server in the `middleware` folder:

```
    └── 5-Express Server
            └── server
                └── controllers
                └── middleware
                    └── headers.js
                    └── validate-session.js <----- ADD THIS
                └── models
            └── client
```

Code

Put the following code inside of `validate-session.js`. There is a little bit of code that is commented out here, leave it like that for now.

```
var jwt = require('jsonwebtoken');
var sequelize = require('../db');
var User = sequelize.import('../models/user');

module.exports = function(req, res, next) {
    // if (req.method == 'OPTIONS') {
    //     next()
    // } else {
        var sessionToken = req.headers.authorization; //1
        console.log(sessionToken) //2
        if (!sessionToken) return res.status(403).send({ auth: false, message: 'No token provided.' }); //3
        else { //4
            jwt.verify(sessionToken, process.env.JWT_SECRET, (err, decoded) => { //5
                if(decoded){
                    User.findOne({where: { id: decoded.id}}).then(user => { //6
                        req.user = user; //7
                        next();
                    },
                    function(){ //8
                        res.status(401).send({error: 'Not authorized'});
                    });
                } else { //9
                    res.status(400).send({error: 'Not authorized'});
                }
            });
        }
    //}
}
```

What Just Happened?

There's a lot here, so take it slow through this explanation. Additional information on the `verify` method can be found **here (https://github.com/auth0/node-jsonwebtoken)** .

1. The variable `sessionToken` is created to hold the token, which is pulled from the authorization header of the request coming in.
2. The token is printed to the console. This is purely for debugging purposes to verify that the token is being sent to the server. It should not be left in the final code, as it is a potential security vulnerability.

3. If no token is present, the `403 Forbidden` error is returned as the response. We have several different error handling responses in this file, so assigning each a different error code or message is a big help in debugging.
4. No `user` property is ever provided in the request, so only tokens will get checked. This prevents unauthorized use of a token that was assigned to a different user.
5. The `verify` method decodes the token with the provided secret, then sends a callback with two variables. If successful, `decoded` will contain the decoded payload; if not, `decoded` remains `undefined`. `err` is `null` by default.
6. If `decoded` has a value, the Sequelize `findOne` method looks for an `id` in the `users` table that matches the `decoded.id` property. This value is then passed into a callback.
7. The callback sets the `user` value for the request as the `id` value passed to it then sends the request on to its next destination. This property will be necessary later in adding to the database.
8. If no matching `id` is found, an error message is thrown.
9. If no value for `decoded`, an error message is thrown.

Before You Move On

Go back and read through all that again. Make a flow chart of what's happening. Explain to a partner what is going on. Do whatever you need to do in order to understand this file. This is some very deep, very detailed code that can be tough to understand. Security should be your #1, #2, and #3 priorities when coding, so it's worth the extra time to get it right.

Changes to App.js

03 - Changes to app.js

In this module, we'll make changes to `app.js` to allow us to create the authenticated routes.

app.js

In the `app.js` file on the server side, make the following changes as noted by the comments:

```
require('dotenv').config();

var express = require('express');
var app = express();
var test = require('./controllers/testcontroller');
var authTest = require('./controllers/authtestcontroller'); //1

var user = require('./controllers/usercontroller');
var sequelize = require('./db');
var bodyParser = require('body-parser');

sequelize.sync(); // tip: {force: true} for resetting tables
app.use(bodyParser.json());
app.use(require('./middleware/headers'));
/******************
 * EXPOSED ROUTES
******************/
app.use('/test', test);
app.use('/api/user', user);

/******************
 * PROTECTED ROUTES
******************/

app.use(require('./middleware/validate-session')); //2
app.use('/authtest', authTest); //3


app.listen(3000, function(){
    console.log('App is listening on 3000.')
});
```

Analysis

In this file, we changed the `app.js` to do the following: 1. We imported the `authtestcontroller` file for access to the endpoints. We will create this file in the next module. Until then, your server might throw an error, as it's looking for a file that doesn't exist. 2. We imported the `validate-session` middleware, which will check to see if the incoming request has a token. 3. Anything beneath the `validate-session` will require a token to access, thus becoming protected. Anything above it will not require a token, remaining unprotected. Therefore, the `test` and `user` routes are not protected, while the `authtest` route is protected.

authtestcontroller.js

04 - authtestcontroller.js

In this module, we'll add a new controller that requires a user token for all requests.

Overview

We are going to add a number of endpoints/routes in this controller. We will give you all of this code and will be analyzing it in the future.

Code

Create an `authtestcontroller.js` file inside of the `controllers` folder, then add the following code:

```
var router = require('express').Router();
var sequelize = require('../db');
var User = sequelize.import('../models/user');
var AuthTestModel = sequelize.import('../models/authtest');

/***********************************
* GET ALL ITEMS FOR INDIVIDUAL USER
***********************************/
router.get('/getall', function (req, res) {
    var userid = req.user.id;

    AuthTestModel
        .findAll({
            where: { owner: userid }
        })
        .then(
            function findAllSuccess(data) {
                res.json(data);
            },
            function findAllError(err) {
                res.send(500, err.message);
            }
        );
});

/***********************************
* POST SINGLE ITEM FOR INDIVIDUAL USER
***********************************/
router.post('/create', function (req, res) {
    var owner = req.user.id;
    var authTestData = req.body.authtestdata.item;

    AuthTestModel
        .create({
            authtestdata: authTestData,
            owner: owner
        })
        .then(
            function createSuccess(authtestdata) {
                res.json({
                    authtestdata: authtestdata
                });
            },
            function createError(err) {
                res.send(500, err.message);
            }
        );
});

/******************
* GET SINGLE ITEM FOR INDIVIDUAL USER
```

```
*****************/
router.get('/:id', function(req, res) {
    var data = req.params.id;
    var userid = req.user.id;

    AuthTestModel
        .findOne({
            where: { id: data, owner: userid }
        }).then(
            function findOneSuccess(data) {
                res.json(data);
            },
            function findOneError(err) {
                res.send(500, err.message);
            }
        );
});

module.exports = router;
```

Short Analysis

Although there are nuances that we'll discuss, the functions should seem somewhat familiar to you based on our previous server functions. Here is a quick explanation for each of them:

Function	Purpose
/getall	Finds all items in the table with the `user id` in the token
/:id	Finds a single item in the the table. Uses both the `id` from the url (primary key) and the `userid` from the token (foreign key).
/create	Adds an item to the table with the `userid` from the token.

Up until now, we've only done `GET` and `POST` requests. A full CRUD (**C**reate **R**ead **U**pdate **D**elete) app lets you update and delete stuff, however, so we need to add some `DELETE` and `UPDATE` functionality. Let's talk a little more about each and set up a route for each before we start testing.

Delete an Item

05 - Delete an Item

Deleting a post isn't as simple as most sites make it seem. Click an `X` or `delete post` and it's magically done, right? If only.

The Code

Add the following function to the bottom of your `authtestcontroller.js` file, right above the export statement:

```
/******************
* DELETE ITEM FOR INDIVIDUAL USER
******************/
        //1               //2
router.delete('/delete/:id', function(req, res) {
    var data = req.params.id; //3
    var userid = req.user.id; //4

    AuthTestModel
        .destroy({ //5
            where: { id: data, owner: userid } //6
        }).then(
            function deleteLogSuccess(data){ //7
                res.send("you removed a log");
            },
            function deleteLogError(err){ //8
                res.send(500, err.message);
            }
        );
});
```

Analysis

What did we just do? 1. When a `DELETE` request is received, the controller looks for a matching function, like what the rest of the HTTP verbs do. 2. We specify what we're doing in our endpoint to make it easy for the user to know what's happening. The `:id` allows a parameter to be passed through the URL to the server so we can use it later. 3. This is the parameter passed through the URL. The same way `req.body` points to the body of the request, `req.params` points to the URL. 4. This is our `userid`, set when `validate-session` is called. 5. `.destroy()` is a Sequelize method to remove an item from a database. See the **Sequelize docs (http://docs.sequelizejs.com/)** for more information. 6. We tell Sequelize what to look for in trying to find an item to delete. If nothing matches exactly, nothing is done. 7. Callback function. This response is sent when the delete is successful. 8. Callback function. This response is sent when the delete is unsuccessful.

Update an Item

06 - Update an Item

Many great apps and sites do not have the ability to update things that have been posted, such as Twitter and Snapchat. These are known as CRD apps. In our case, we want a full CRUD app, so let's get to work. This is the last major addition we need for our server to be complete.

The Code

Add the following at the bottom of `authtestcontroller.js`, right above the export statement.

```
/******************
* UPDATE ITEM FOR INDIVIDUAL USER
******************/
       //1    //2
router.put('/update/:id', function(req, res) {
    var data = req.params.id; //3
    var authtestdata = req.body.authtestdata.item; //4

    AuthTestModel
        .update({ //5
            authtestdata: authtestdata //6
        },
        {where: {id: data}} //7
        ).then(
            function updateSuccess(updatedLog) { //8
                res.json({
                    authtestdata: authtestdata
                });
            },
            function updateError(err){ //9
                res.send(500, err.message);
            }
        )
});
```

Analysis

1. `PUT` is one of the HTTP verbs that has to be weird by not telling you what it does. `PUT` replaces whatever is already there with what we give it. In other words, `PUT` means update.
2. To make it easier on the user, we use `update` in our route. We also allow a variable (`id`) to be passed through the URL again.
3. The parameter taken from the URL.
4. Our data we want to put into the database, replacing what already exists.
5. `update` is a Sequelize method which takes two arguments.
6. First argument of `update`. Contains an object holding the new value we want to edit into the database.
7. Second argument of `update`. Tells Sequelize where to place the new data if a match is found.
8. Callback function. Runs if update is successful, and returns the data entered.
9. Callback function. Runs if update is not successful, and returns the error message.

And that's it! Our server is done! Let's do some final testing with Postman, and then move on to setting up our client to navigate these authenticated routes.

Postman Testing

07 - Postman Testing

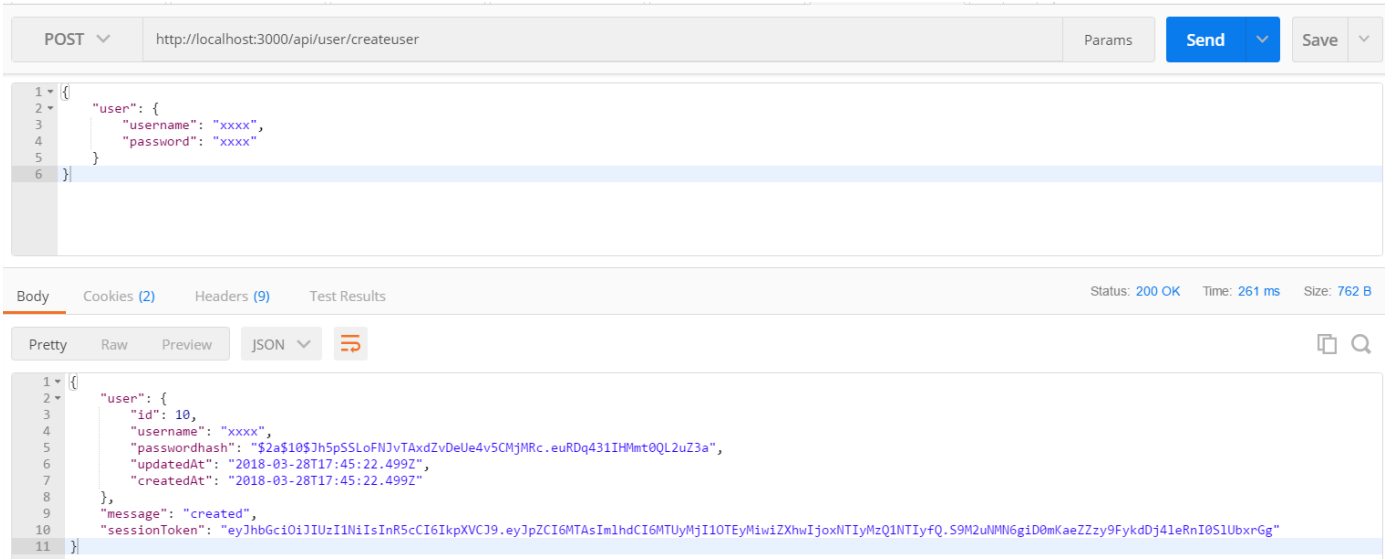
In this module, we'll quickly test our routes with Postman. We want to run 8 tests between the user and authtest controllers:

1. Create a new user.
2. Get a new token.
3. Set up the headers with the proper token.
4. Create an item with a specific user.
5. Get all items for a specific user.
6. Get a single item for a specific user.
7. Update an item for a specific user.
8. Delete an item for a specific user.

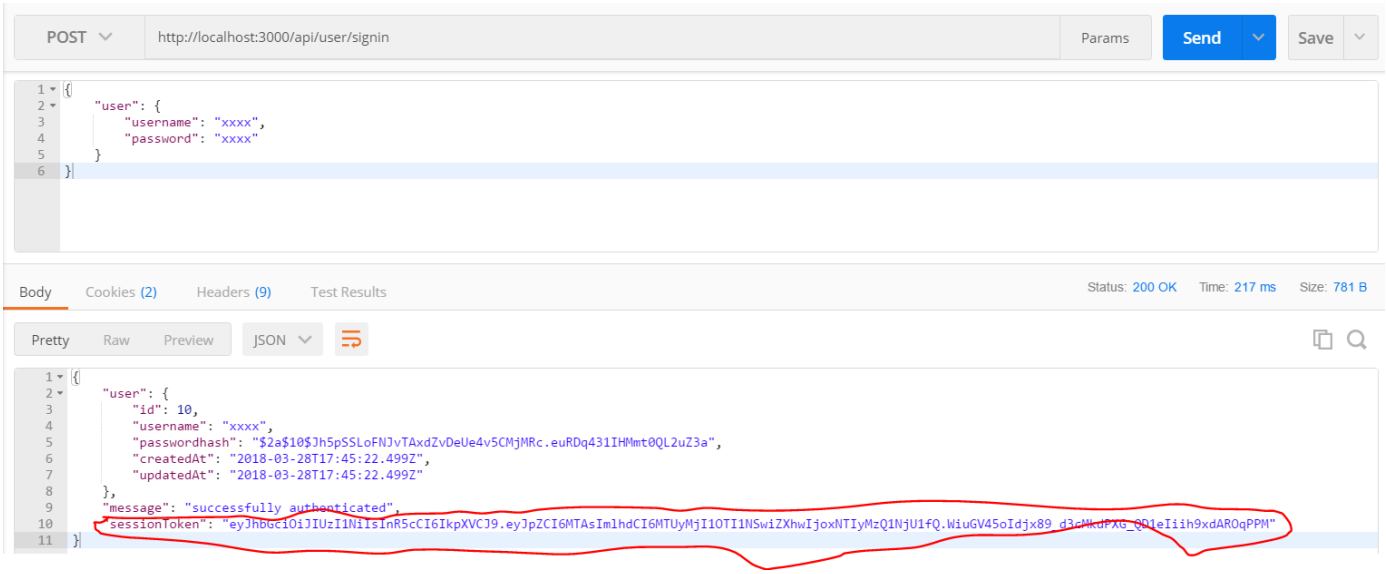
After each step, check the relevant table in your database to make sure that everything worked properly. There are screenshots below to show you how each request should look, but try to do them by yourself first.

Screenshots fot Tests:

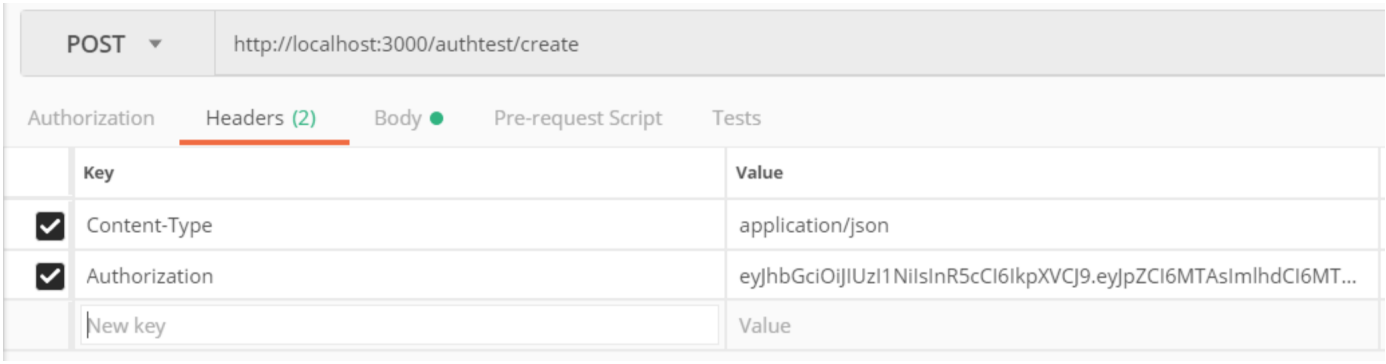
Test your Endpoints: 1. Create a new user.



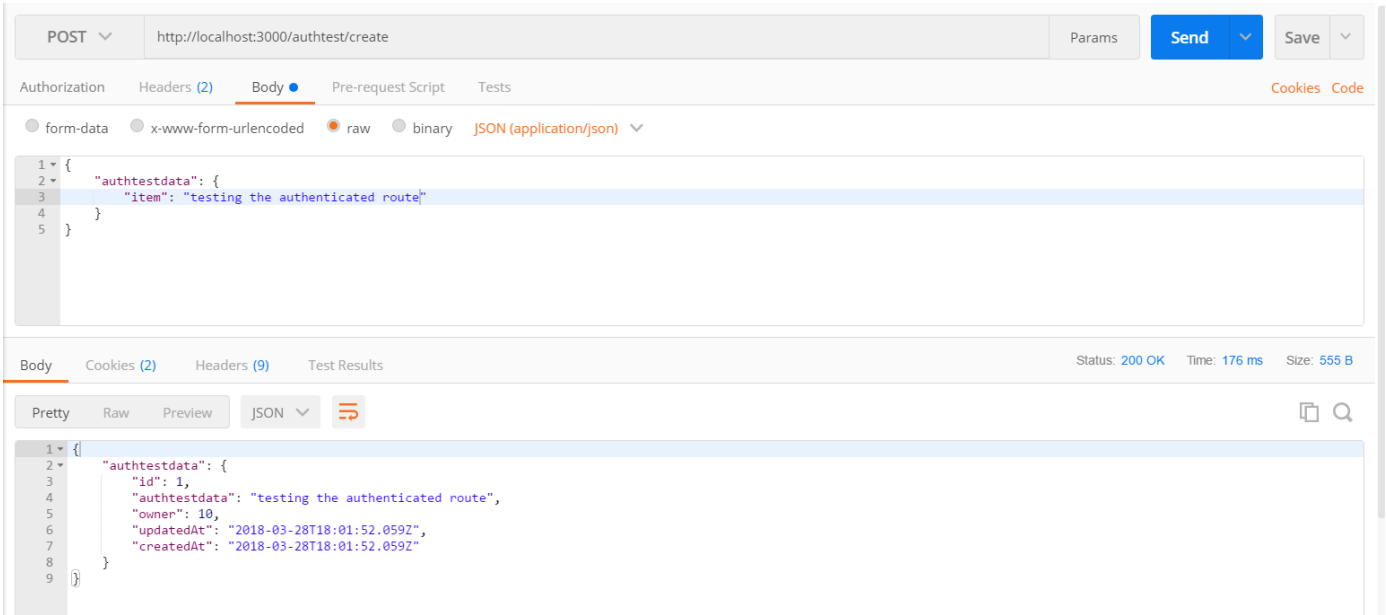
1. Get a new token.



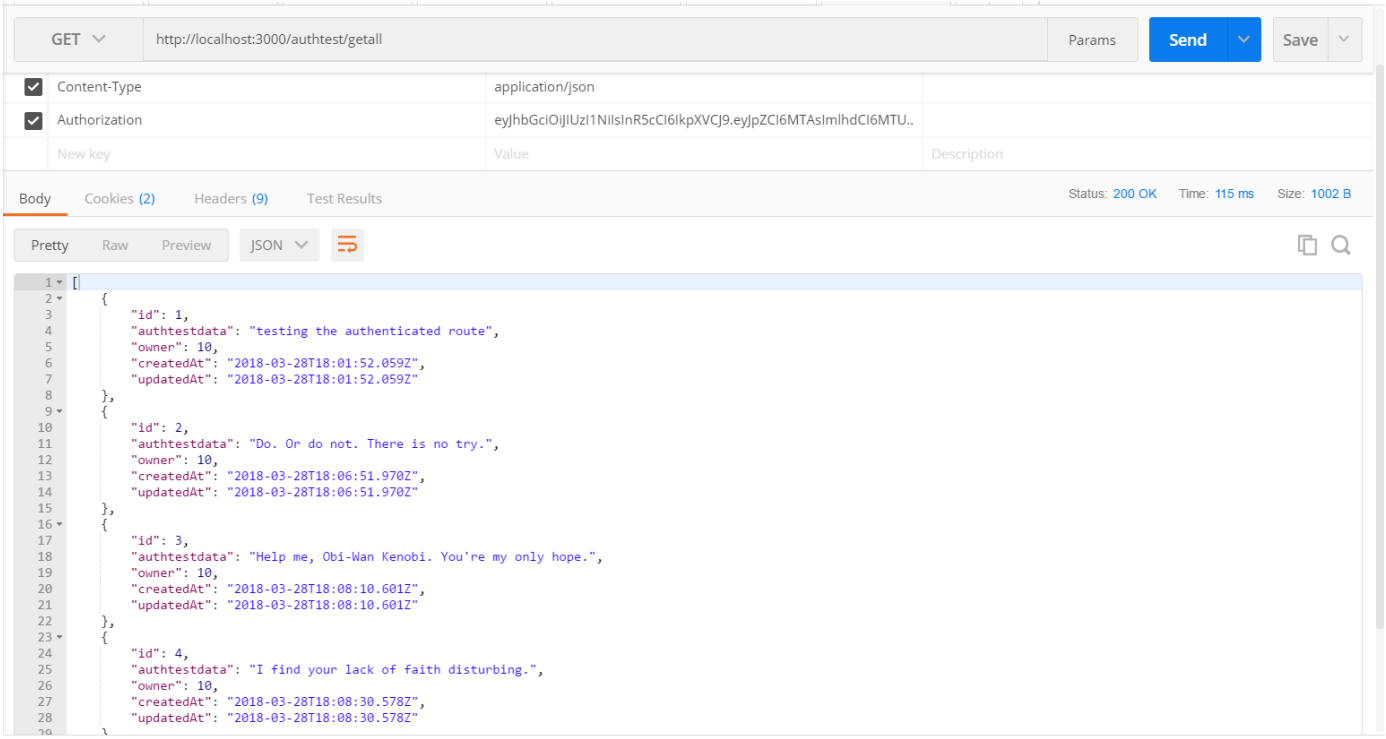
1. Set up the headers with the proper token. Just copy and paste the token you just got. And our application is returning json.



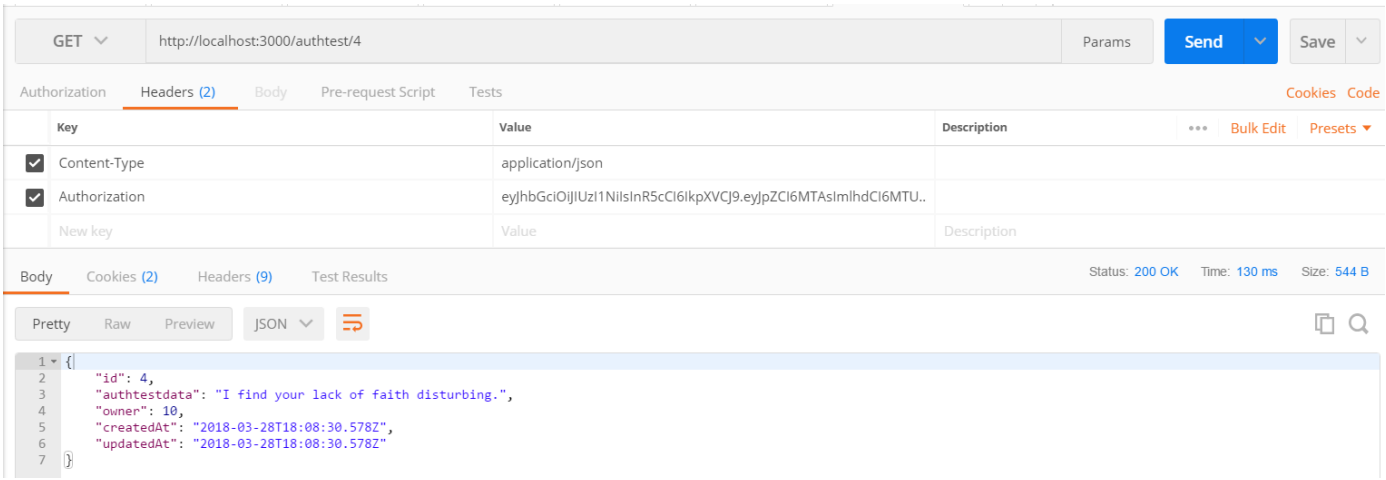
1. Create an item with a specific user `authtest/create`.



1. Get all items for a specific user `authtest/getall`. We added a few items.



1. Get a single item for a specific user `authtest/id`.



1. Update an item for a specific user `authtest/update/id`. If you have more than one item, the updated one will move to the bottom of the list.

GET http://localhost:3000/authtest/getall Params Send Save

Body Cookies (2) Headers (9) Test Results Status: 200 OK Time: 39 ms Size: 1.25 KB

Pretty Raw Preview JSON


```
    },
    {
        "id": 2,
        "authtestdata": "Do. Or do not. There is no try.",
        "owner": 10,
        "createdAt": "2018-03-28T18:06:51.970Z",
        "updatedAt": "2018-03-28T18:06:51.970Z"
    },
    {
        "id": 3,
        "authtestdata": "Help me, Obi-Wan Kenobi. You're my only hope.",
        "owner": 10,
        "createdAt": "2018-03-28T18:08:10.601Z",
        "updatedAt": "2018-03-28T18:08:10.601Z"
    },
    {
        "id": 4,
        "authtestdata": "I find your lack of faith disturbing.",
        "owner": 10,
        "createdAt": "2018-03-28T18:08:30.578Z",
        "updatedAt": "2018-03-28T18:08:30.578Z"
    },
    {
        "id": 5,
        "authtestdata": "Luke, I am your father.",
        "owner": 10,
        "createdAt": "2018-03-28T18:12:01.890Z",
        "updatedAt": "2018-03-28T18:12:01.890Z"
    },
    {
        "id": 6,
        "authtestdata": "Meesa Jar Jar Binks.",
        "owner": 10,
        "createdAt": "2018-03-28T18:17:49.058Z",
        "updatedAt": "2018-03-28T18:17:49.058Z"
    }
```


PUT http://localhost:3000/authtest/update/5 Params Send Save

Authorization Headers (2) Body Pre-request Script Tests Cookies Code

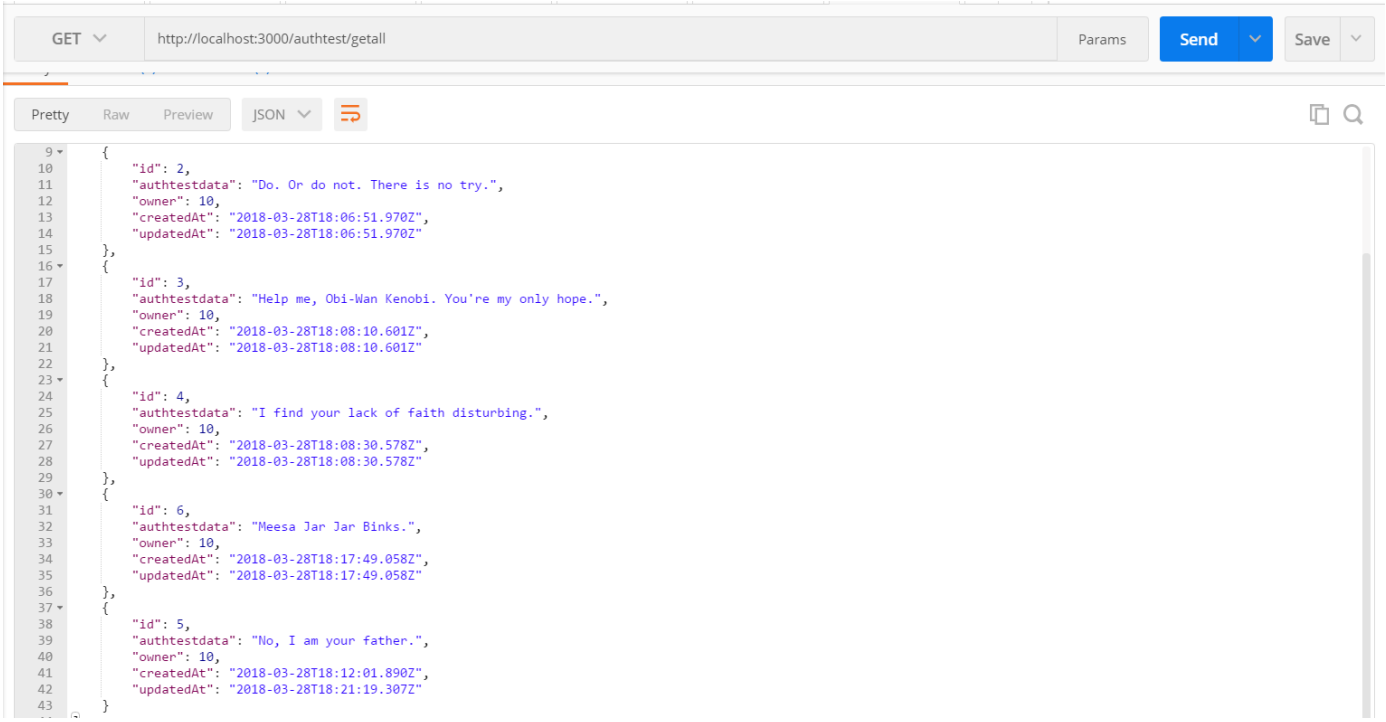
form-data x-www-form-urlencoded raw binary JSON (application/json)

```
{
    "authtestdata": {
        "item": "No, I am your father."
    }
}
```

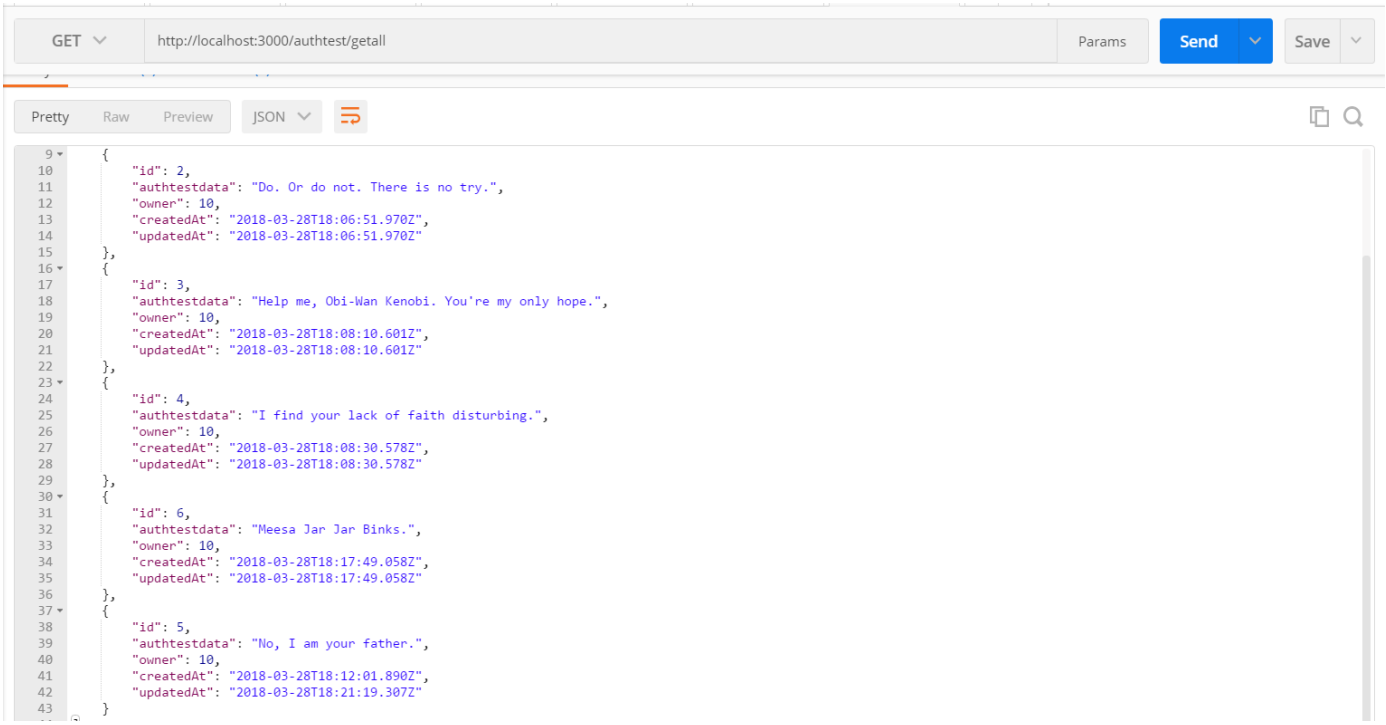
Body Cookies (2) Headers (9) Test Results Status: 200 OK Time: 105 ms Size: 431 B

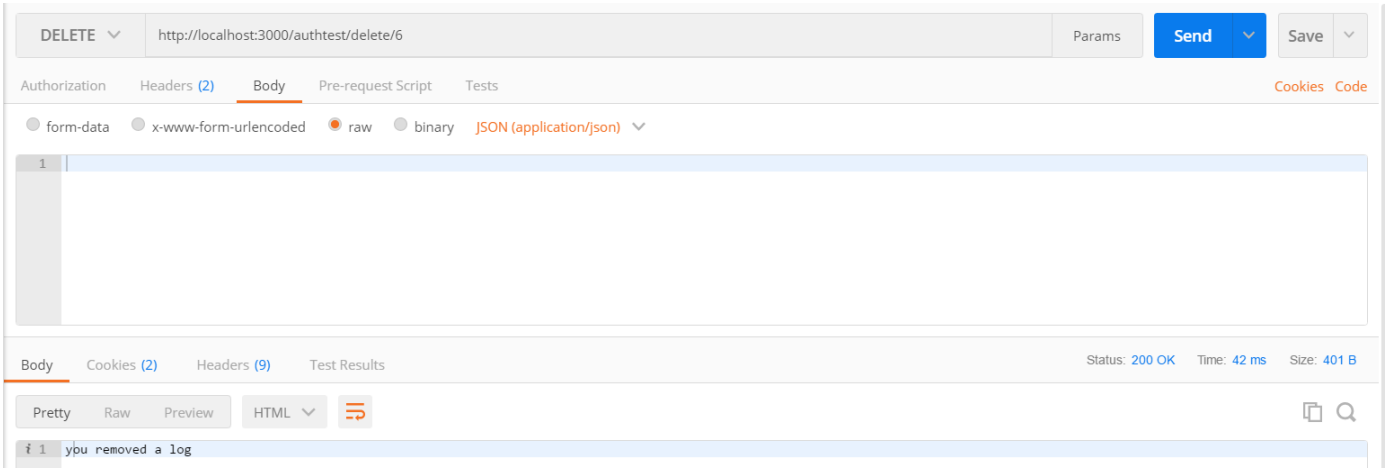
Pretty Raw Preview JSON

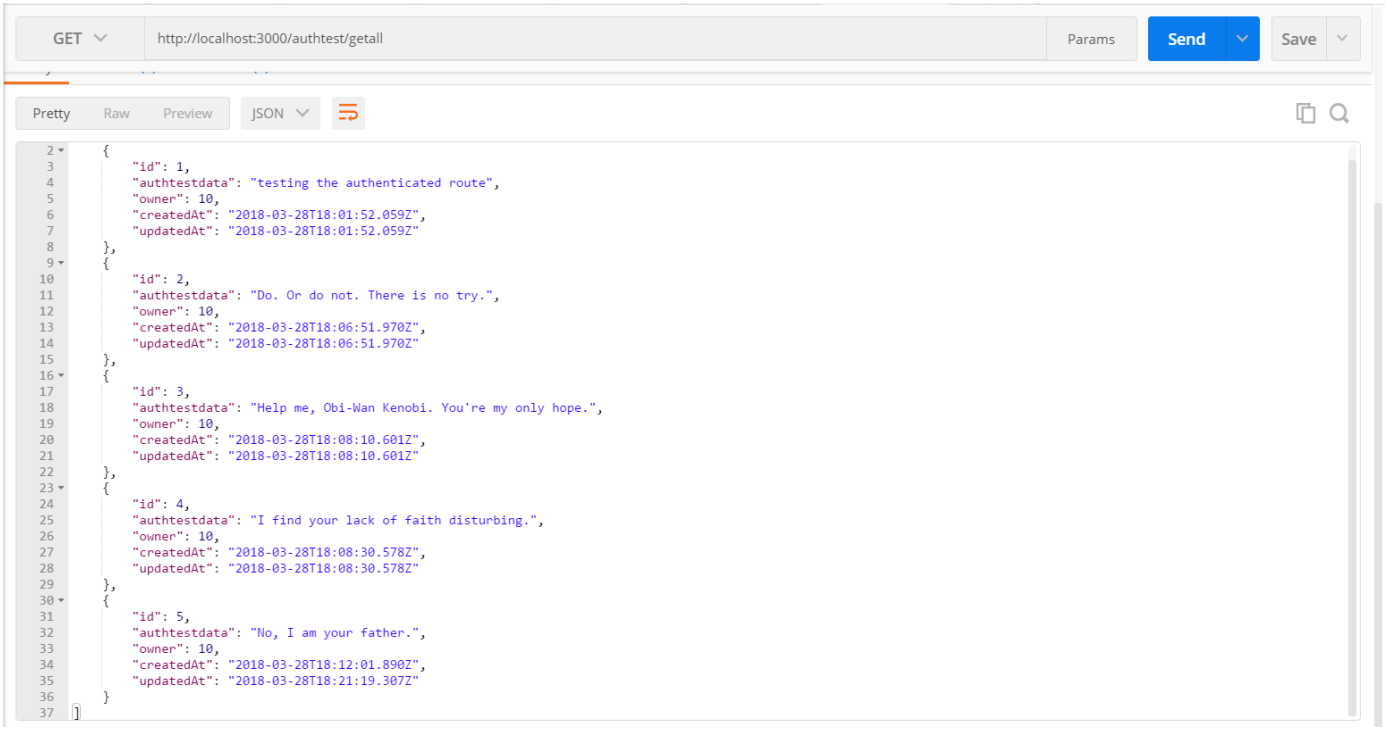

```
{
    "authtestdata": "No, I am your father."
}
```



Had to get the quote right obviously. 8. Delete an item for a specific user `authtest/delete/id`.







It's like the Gungans never existed.

Additions to Index

00 - Additions to index

Before we get started working on our client on making authenticated requests to our authenticated routes, we'll add more code to our HTML file.

Code

Add the following code to our current `index.html` file. NOTE: You will be adding to the existing table in the file. These are more rows for that table. You will add this code under the closing `</tr>` tag in the 5th row. Please copy and paste the following code for now:

```
<!--PASTE THIS CODE AFTER ROW 5-->
                <tr>
                    <th scope="row">6</th>
                    <td>POST Sign Up
                        <code>/api/user/createuser</code>
                    </td>
                    <td>
                        <span>
                        <input type="text" id="userSignUp" placeholder="Username" />
                        </span>
                        <span>
                        <input type="password" id="passSignUp" placeholder="Password" />
                        </span>
                        <span>
                        <button onclick="userSignUp();">Submit</button>
                        </span>
                    </td>
                    <td>
                        See Console
                    </td>
                </tr>
                <tr>
                    <th scope="row">7</th>
                    <td>POST Sign In
                        <code>/api/user/signin</code>
                    </td>
                    <td>
                        <span>
                        <input type="text" id="userSignin" placeholder="Username" />
                        </span>
                        <span>
                        <input type="password" id="passSignin" placeholder="Password" />
                        </span>
                        <span>
                        <button onclick="userSignIn();">Submit</button>
                        </span>
                    </td>
                    <td>
                        See Console
                    </td>
                </tr>
                <tr>
                    <th scope="row">8</th>
                    <td>Function: Get Session Token</td>
                    <td>
                        <button onclick="getSessionToken();">Print Token</button>
                    </td>
                    <td>
                        See Console
                    </td>
                </tr>
                <tr>
                    <th scope="row">9</th>
                    <td>Authenticated Request
```

```
            <code>/authtest/getall</code>
        </td>
        <td>
            <button onclick="fetchAllFromAuthRoute();">GET</button>
        </td>
        <td>
            See Console
        </td>
    </tr>
    <tr>
        <th scope="row">10</th>
        <td>Authenticated POST Request to <code>/authtest/create</code></td>
        <td>
            <input type="text" id="authTestData" placeholder="Enter Data"/>
            <button onclick="postToAuthRouteCreate();">POST</button>
        </td>
        <td>
            See Console
        </td>
    </tr>
    <tr>
        <th scope="row">11</th>
        <td>GET Single Item
            <code>/authtest/id</code>
        </td>
        <td>
            <input type="text" id="getNumber" placeholder="Post ID #"/>
            <button onclick="getOneByUser();">Get Single Item</button>
        </td>
        <td>
            <label id="getItemValue"></label>
        </td>
    </tr>
    <tr>
        <th scope="row">12</th>
        <td>Update Single Item
            <code>/authtest/update/id</code>
        </td>
        <td>
            <input type="text" id="updateNumber" placeholder="Post ID #"/>
            <input type="text" id="updateValue" placeholder="New Data"/>
            <button onclick="updateItem();">Update Item</button>
        </td>
        <td>
            <label id="newItemValue"></label>
        </td>
    </tr>

    <tr>
        <th scope="row">13</th>
        <td>Delete Single Item
            <code>/authtest/delete/id</code>
        </td>
        <td>
            <input type="text" id="deleteNumber" placeholder="Post ID #"/>
            <button onclick="deleteItem();">Delete Single Item</button>
        </td>
        <td>
            See Console
        </td>
    </tr>
    <tr>
        <th scope="row">14</th>
        <td>Delete Single Item From DOM</td>
        <td>
            <button onclick="fetchFromOneDisplayData();">Display data</button>
        </td>
        <td>
            <ul id="fourteen">
            </ul>
        </td>
    </tr>
```

Analysis

Take a minute to check through the additions. Notice that we've added the following: 1. 8 more table rows 2. Buttons, lists, labels, and input fields added to various rows. 3. Classes and ids in elements for future use.

1. Names of functions that we will soon create.

Test

You should see something similar to the following when you run the application:

4	POST Data `/test/seven`	Post	• Posted data: • At:
5	GET and display json `/test/one`	Display data	
6	POST Sign Up `/authtest/createuser`	Username Password Submit	See Console
7	Function: Get Session Token	Print Token	See Console
8	Authenticated Request `/authtest/getall`	GET	See Console
9	Authenticated POST Request to /authtest/create	Enter Data POST	See Console
10	GET Single Item `/authtest/id`	Post ID # Get Single Item	
11	Update Single Item `/authtest/update/id`	Post ID # New Data Update Item	
12	Delete Single Item `/authtest/delete/id`	Post ID # Delete Single Item	See Console
13	Delete Single Item	Display data	

Anatomy of a Request

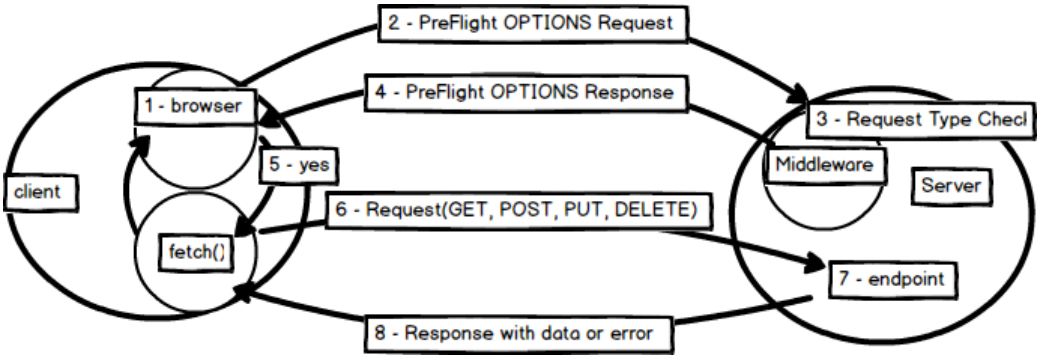
01 - Anatomy of a Request

In this module, we'll go deeper into discussing the request/response lifecycle and examine how requests interact with our middleware and server.

Anatomy of the Request

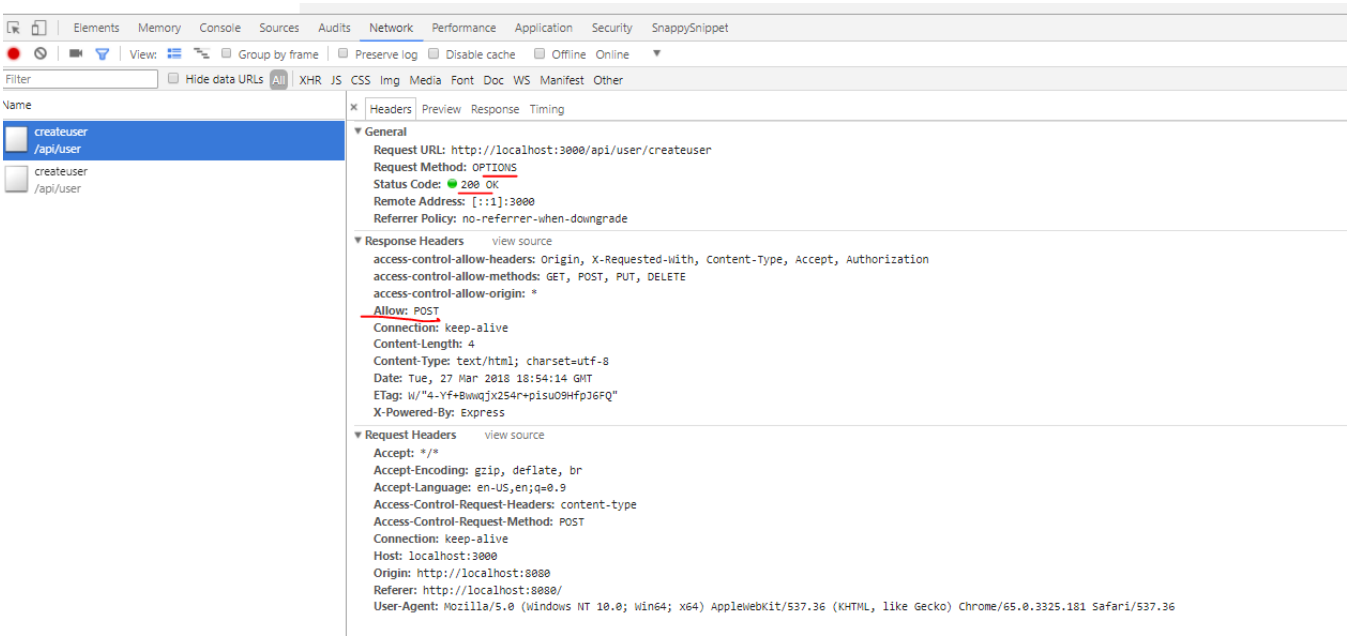
It's important to note that there is a lot going on under the hood with requests in general. This complexity is in part to adhere to HTTP protocols and to manage security and traffic between domains. Let's dig in and look underneath the surface a little bit.

When we complete a `fetch` here, we are kicking off a multi-step process in this request. It goes something like this:

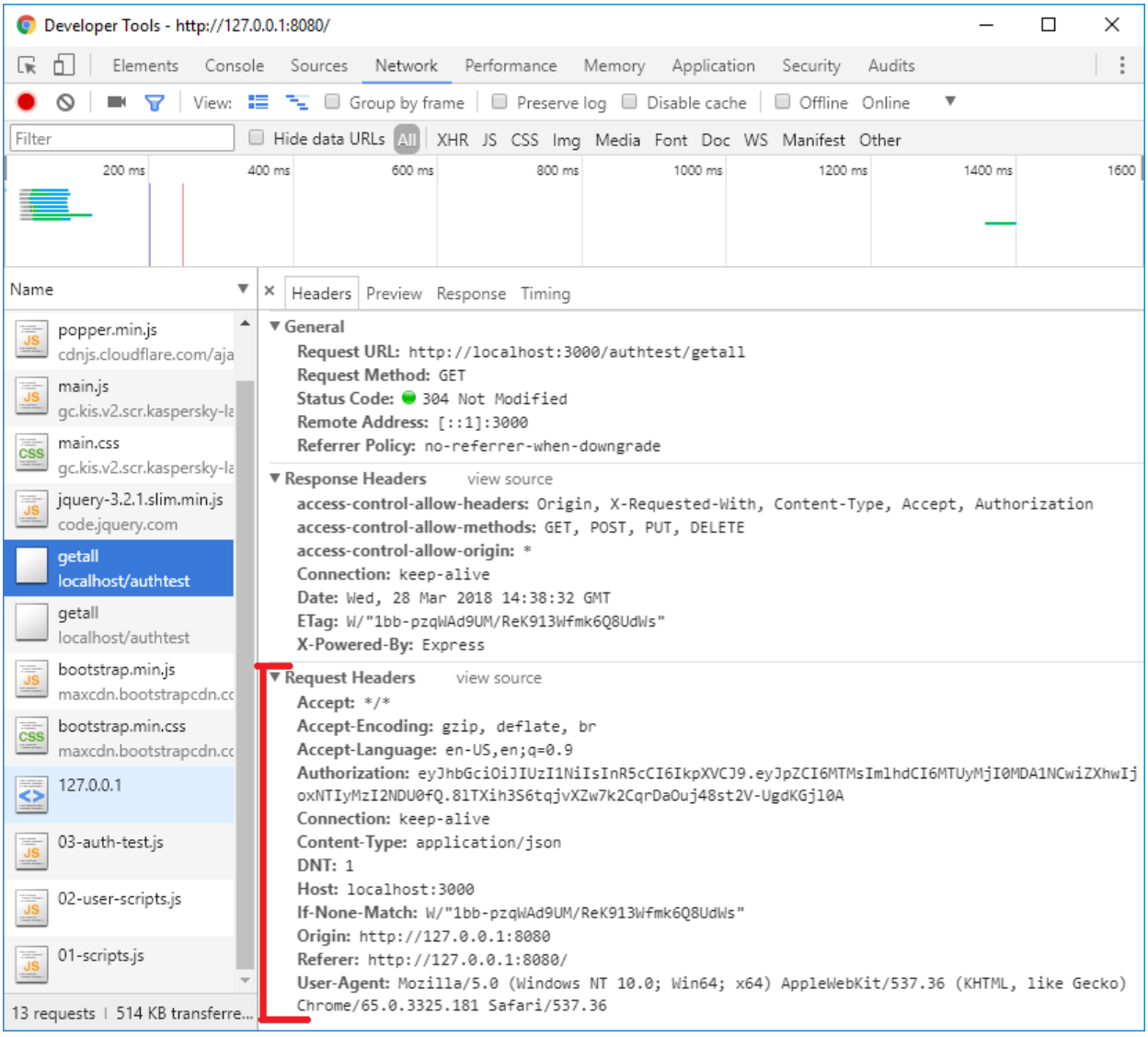


Analysis

1. When the method fires off, `fetch()` notifies the browser to send a Pre-Flight from `localhost:8080`.
2. The browser fires off an `OPTIONS` request. `OPTIONS` is an HTTP verb, like `GET`, `POST`, `PUT`, `DELETE`. Check on the Postman list; you'll see it there. The `OPTIONS` verb allows the client to determine the options associated with our server without having to dig in and do data retrieval or deal with resources in the server. It's an intermediary between domains that says, "Hey, will we be able to do this here if we come in?"
3. The `OPTIONS` HTTP protocol checks in with the middleware on the server for the request type. Essentially if we fire off a `GET` request for a certain route from the client, the browser does an initial scan and checks to be sure that the type of request can happen.
4. If the request type is enabled in the server, specifically the headers, the Pre-Flight `OPTIONS` response is sent back with the listing of that request type.
5. If the Pre-Flight `OPTIONS` request determines that the request is allowed, the `fetch()` method fires off and the second request for data can be made. Notice the `Allow: POST` in the Response. These images are just for display purposes only. You don't need to do anything right now.

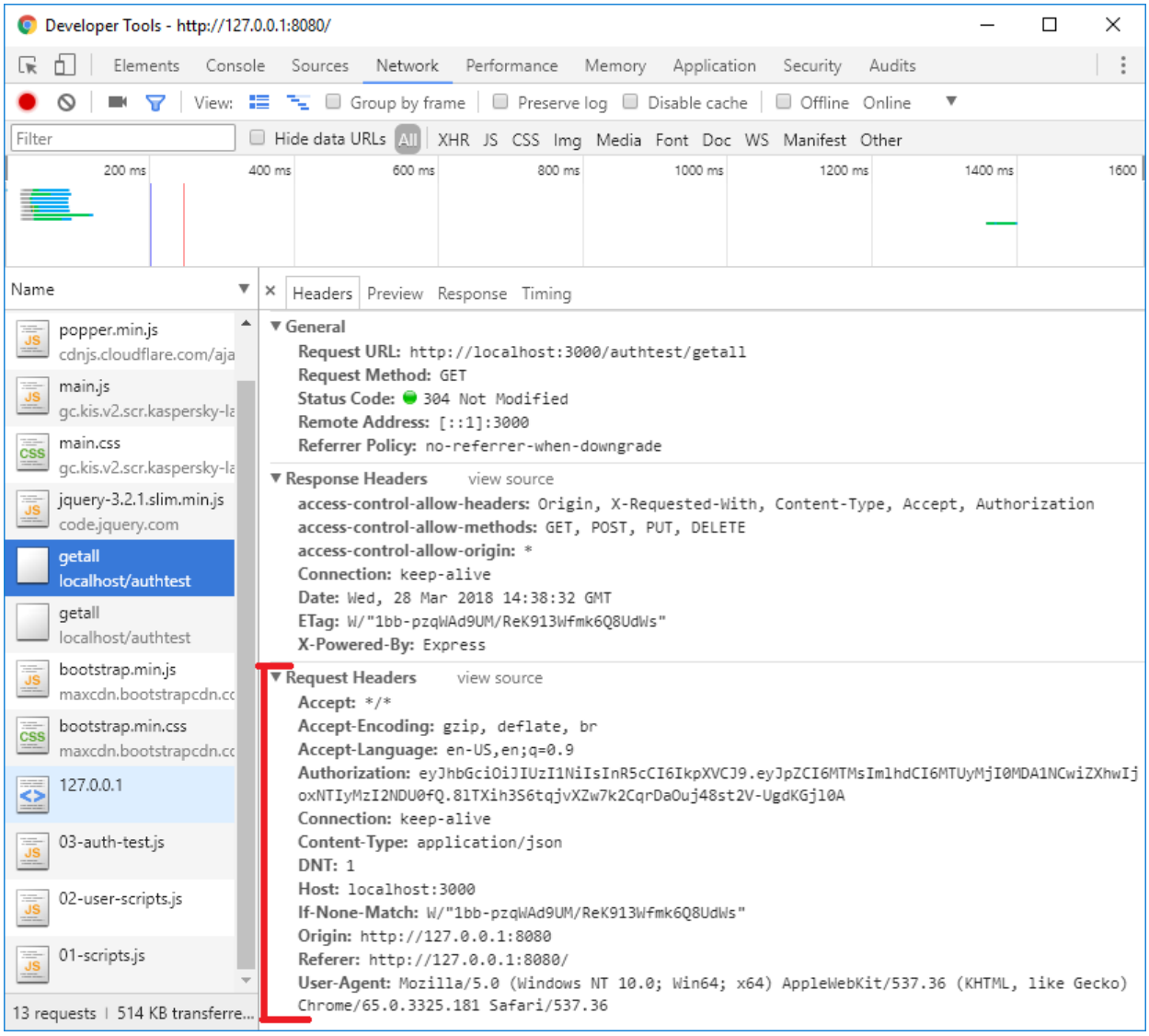


6. The `fetch()` script fires off the second request approved by the Pre-Flight request.
7. The second request makes it to the server endpoint, which processes the request.
8. The server sends back a new response.



The Important Points:

1. Before it sends off the `GET` or `POST` request to the server, the browser performs a `pre-flight` check.
2. If everything checks out with our Pre-Flight request, the server responds with a `200 OK` and `fetch()` follows with the request/response that has been proclaimed (`GET`, `POST`, `PUT`, etc.), as seen here:



Create User

02 - Create User

In this module, we'll add client methods for creating and logging in as a user.

User Scripts

Please add a `02-user-scripts.js` file inside of the client folder:

```
    └── 5-Express Server
            └── server
            └── client
                └── 01-scripts.js
                └── 02-user-scripts.js
                └── index.html
```

We'll add some of our auth logic in this file.

Script Tags

Follow these steps to wire up the `js` file:

1. Go to `index.html`.
2. Go to the bottom of that same file, underneath the Bootstrap scripts and the `01-scripts.js` file, and add the `02-user-scripts.js` tag like this:

```
    <script src="01-scripts.js"></script>
    <script src="02-user-scripts.js"></script>

</body>

</html>
```

userSignUp

At the top of this new script file,`02-user-scripts.js`, let's add the code for signing up a user:

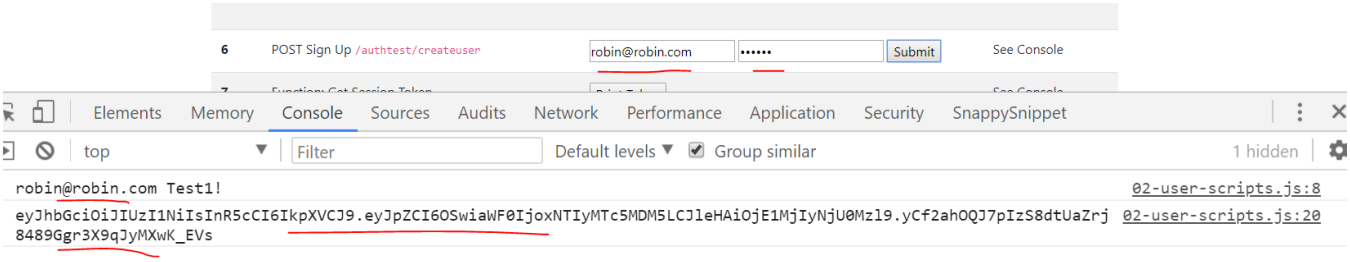
```
/********************
 * POST - /createuser
********************/
function userSignUp(){
    let userName = document.getElementById('userSignUp').value; //1
    let userPass = document.getElementById('passSignUp').value;
    console.log(userName, userPass);

    let newUserData = {user : { username: userName, password: userPass}}; //2
    fetch('http://localhost:3000/api/user/createuser', {
        method: 'post',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(newUserData) //3
    })
    .then(response => response.json())
    .then(function (response) {
        console.log(response.sessionToken);
        let token = response.sessionToken; //4
        localStorage.setItem('SessionToken', token);  //5
    });
}
```

Analysis

1. Here, we grab the value of the user/password data from the `createuser` input field in the `index.html` file.
2. The variables used to store the sign up info from the DOM get passed into the values of the `username` and `password` properties. We package everything up in a user object.
3. In the request object in our `fetch()` call, we pass in the `newUserData` variable to be sent off in the body of our request to the server.
4. We get the `sessionToken` from the response and store it in a `token` variable.
5. In our `localStorage`, we call the `setItem` function and store the token in `localStorage`. This will keep our token safely stored in our local window.

Test

1. Make sure the client and server are running.
2. Open up the client in the browser. Go to **Step #6: POST Sign Up** on the table.
3. Sign up with a new user and press send. You should see the following:



4. Notice that we have the username, password, and a token printing in the console window.
5. You should also crack open Postgres, refresh your database, and look at the User table. You should see the new user added there now.

userSignIn

The `signin` function is exactly the same as `signup`. We're just pulling from different fields and sending a request to a different endpoint. Put this code under `userSignUp`:

```
function userSignIn(){
    let userName = document.getElementById('userSignin').value;
    let userPass = document.getElementById('passSignin').value;
    console.log(userName, userPass);

    let userData = {user : { username: userName, password: userPass}};
    fetch('http://localhost:3000/api/user/signin', { //<---signin route used
        method: 'post',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(userData)
    })
    .then(response => response.json())
    .then(function (response) {
        console.log(response.sessionToken);
        let token = response.sessionToken;
        localStorage.setItem('SessionToken', token);
    });
}
```

Testing

Follow the `signup` instructions to create a couple more users. Then use `Step 7: POST SignIn` to sign in as each user. Notice that each one gets a different session token, which is reset in `localStorage` each time.

Getting a Token

03 - Getting a Token

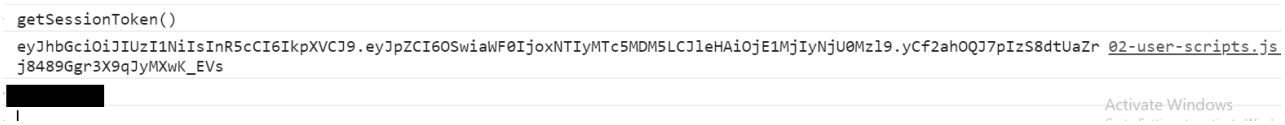
Let's just create a function that will get our token and print it in the console.

Code

It is often handy to have a function that retrieves the session token. Let's write a basic token. You can add this directly under our last function in `02-user-scripts.js`:

```
/*****************************
 * HELPER FUNCTION FOR TOKEN
*****************************/
function getSessionToken(){
    var data = localStorage.getItem('SessionToken');
    console.log(data);
    return data;
}
```

Save and run the app. If you call the function in the console, it will print:



This is handy for passing the token around in our app, but it's also handy to access the token for testing purposes.

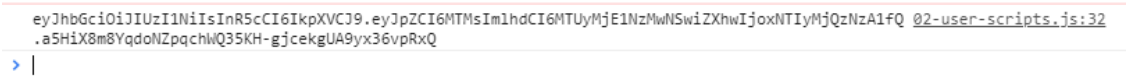
Another Way

Another way to access your token is simply by calling the `localStorage.getItem("SessionToken")` method:

Test

Note that we've made a spot in the DOM for the `getSessionToken()` method. Let's test that:

1. Make sure that both the server and client are running.
2. Open the console.
3. Go to **Step #8: Function: Get Session Token** on the table.
4. Click the button to print the token to the console:



Get Items from One User

04 - Get Items From One User

In this module, we'll add a client method for making an authenticated request with a token to an authenticated route.

Authenticated Request Defined

Authentication is the process of identifying whether a client is eligible to access a resource. An authenticated request usually means that a client has some token or cookie allowing access to a resource. Hence, for clarity in this book, we'll refer to Authenticated Requests as being synonmous with a user that has a token. Think of it as a user that is logged in.

Auth Test

Let's add a `03-auth-test.js` file inside of the client folder:

```
    └── 5-Express Server
            └── server
            └── client
                └── 01-scripts.js
                └── 02-user-scripts.js
                └── 03-auth-test.js
                └── index.html
```

We'll add all of our authenticated request logic in there.

We'll also need to add the script tag to the bottom of the `index.html` file:

```
    <script src="01-scripts.js"></script>
    <script src="02-user-scripts.js"></script>
    <script src="03-auth-test.js"></script>

</body>

</html>
```

Code

Add the following code to `03-auth-test.js`:

```
function fetchAllFromAuthRoute() {
    const fetch_url = `http://localhost:3000/authtest/getall`
    const accessToken = localStorage.getItem('SessionToken') //1

    const response = fetch(fetch_url, {
        method: 'GET', //2
        headers: {
            'Content-Type': 'application/json', //3
            'Authorization': accessToken //4
        }
    })
        .then(response => {
            return response.json();
        })
        .then(data => {

            console.log(data)
        })
}
```

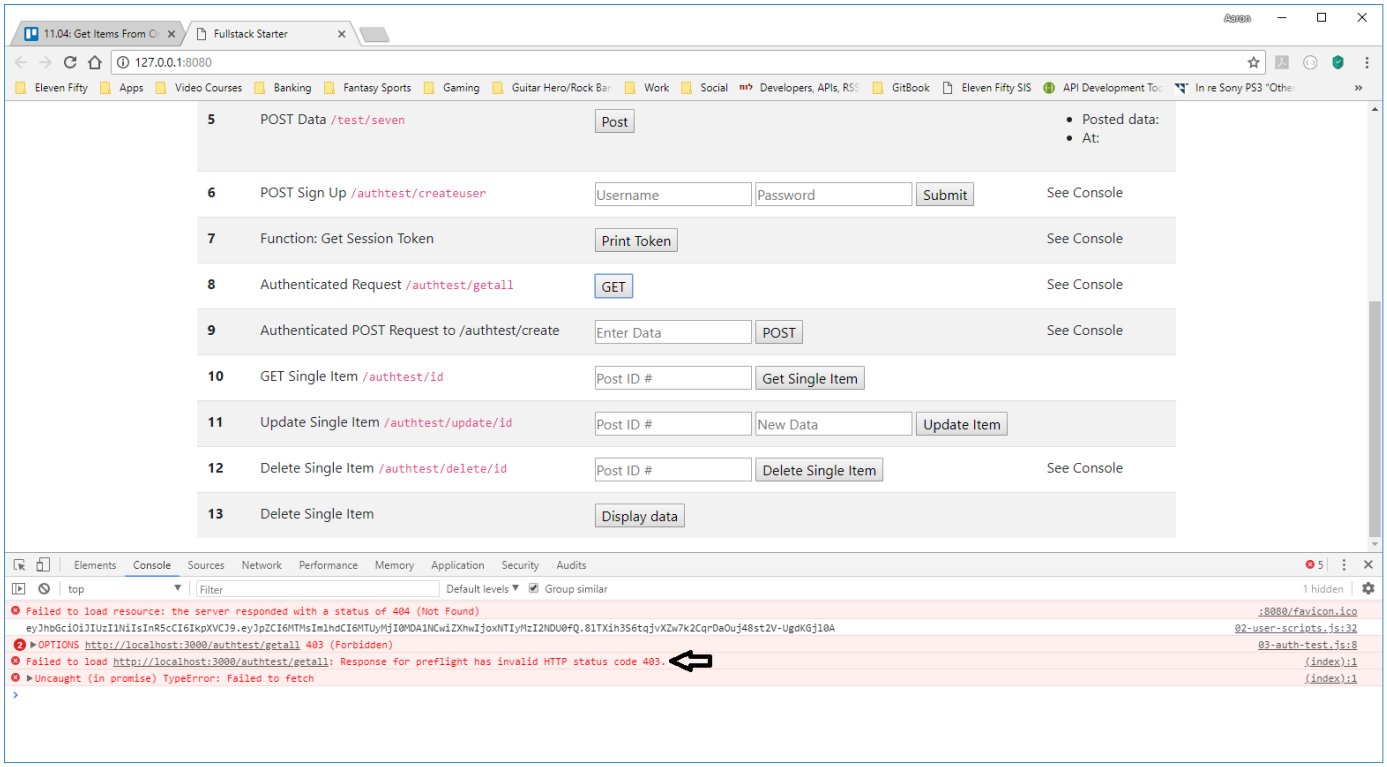
Analysis

1. Since we stored our token in `localStorage`, we can access it by using the `getItem` method to get it back from localStorage and put it in a variable. Note that we could also use our `getSessionToken()` method for this task.
2. By default, `fetch` runs a `GET` request. We can use the `method` property to send other requests. In this case, we're still sending a `GET`.
3. The `Content-Type` header tells the server what kind of data is being sent in our PreFlight request, if any.
4. The `Authorization` header provides some sort of encrypted data allowing access to the server, in this case our token.

Test

1. Make sure both the server and client are running.
2. Open the console.
3. Go to Step 9.
4. Press the button.

You should see an error similar to the following image:



The problem isn't with our code here. The problem lies with a file on the server side: the `validate-session.js` file, and specifically how that file handles the pre-flight `OPTIONS` request sent by our browser.

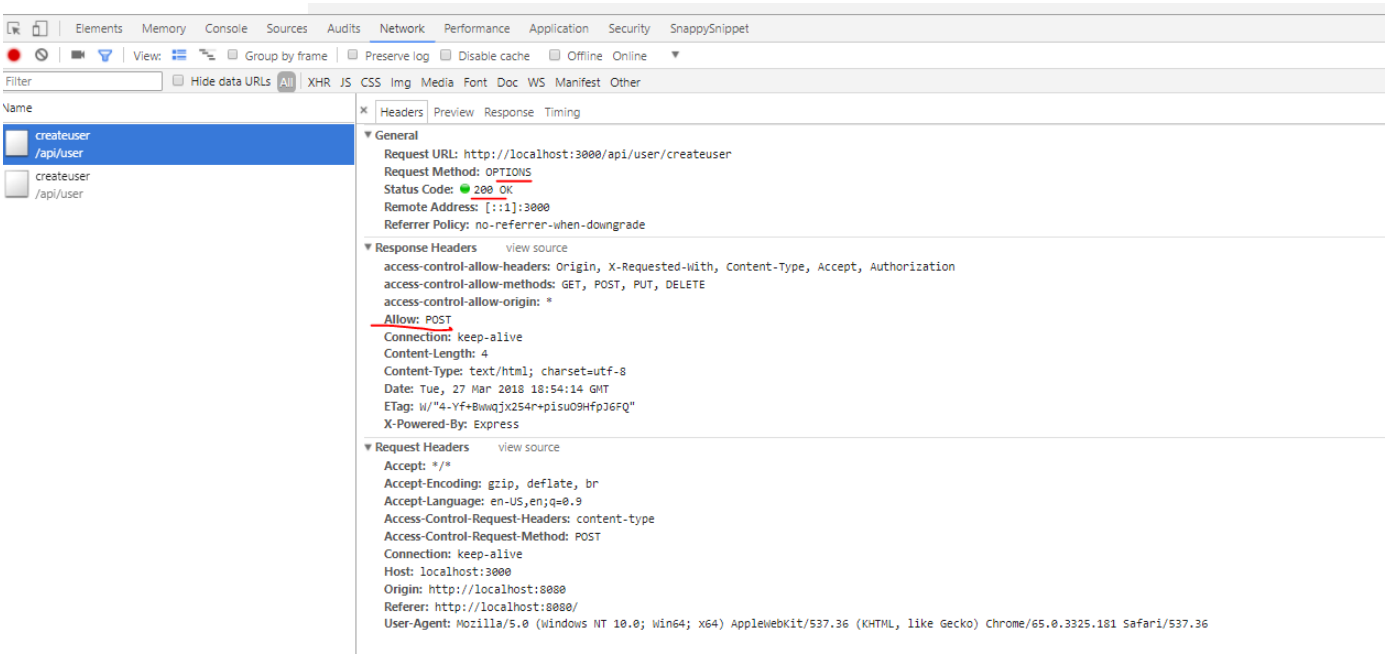
Small Refractor to `validate-session.js`

As a reminder, here is the `validate-session` function:

```
module.exports = function(req, res, next) {
    // if (req.method == 'OPTIONS') {
    //     next()
    // } else {
        var sessionToken = req.headers.authorization; //PROBLEM IS RIGHT HERE
        console.log(sessionToken)
        if (!sessionToken) return res.status(403).send({ auth: false, message: 'No token provided.' });
        else {
            jwt.verify(sessionToken, process.env.JWT_SECRET, (err, decoded) => {
                if(decoded){
                    User.findOne({where: { id: decoded.id}}).then(user => {
                        req.user = user;
                        next();
                    },
                    function(){
```

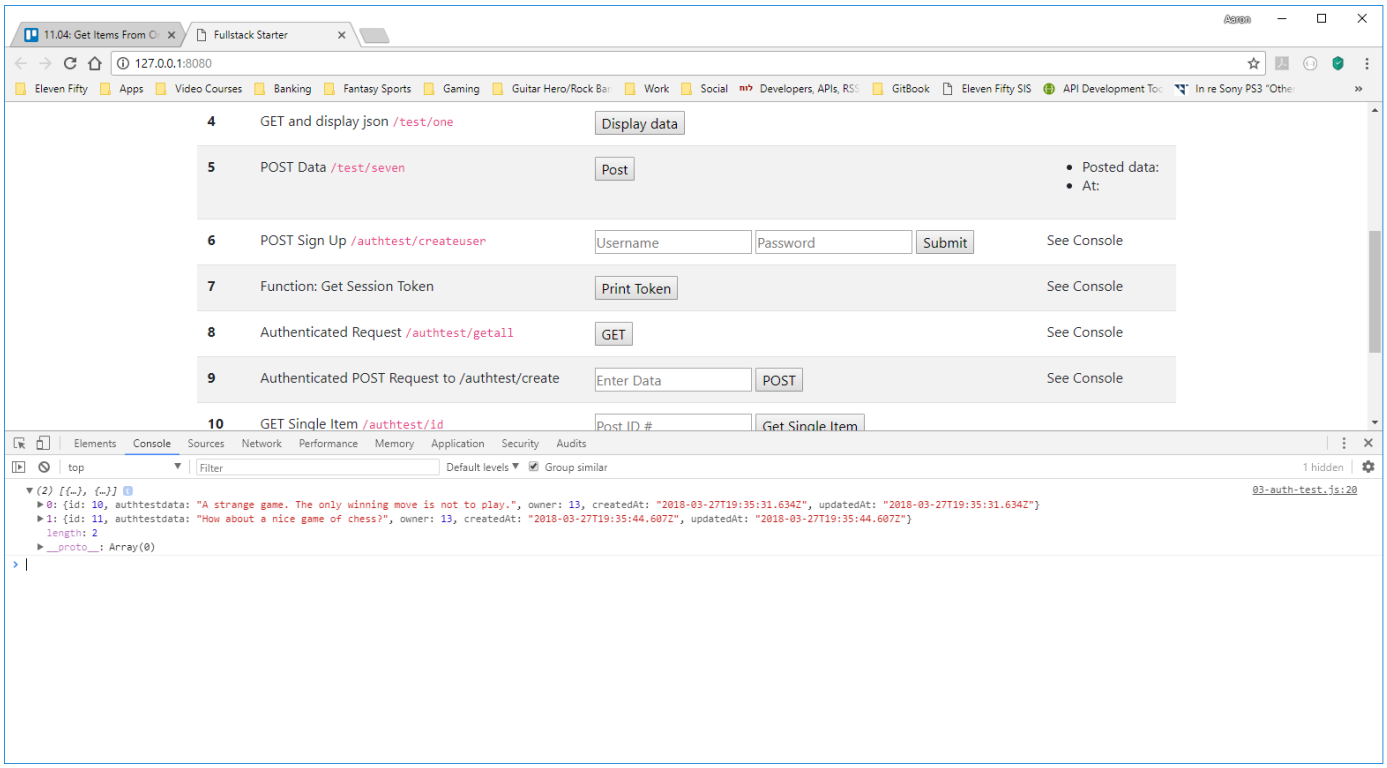
```
                    res.status(401).send({error: 'Not authorized'});
                });
            } else {
                res.status(400).send({error: 'Not authorized'});
            }
        });
    }
  //}
}
```

Notice the parts that are commented out: the `if` statement at the top and the corresponding `else`. When we used Postman to test, we never sent an `OPTIONS` request. Here is the result of that request:



You can see that there isn't an `Authorization` header on that request, so when `validate-session` looks for `req.headers.authorization`, it comes back undefined, breaking the rest of the function. That's where this conditional comes into play. One of the properties on `fetch` is `method`; This is where we tell fetch what type HTTP request to send (`GET`, `POST`, etc.). This conditional allows us to tell the program to let any request where `req.method` is `OPTIONS` through without checking for a session token. This way the pre-flight check can occur, then the program will look for and verify a token on any other request.

Un-comment the `if/else` statement at the top, as well as the closing curly bracket at the bottom, then run the test above again. It should go through this time. However, since you've just created your user, you will probably see an empty array. This is coming from the Postgres table:


11.04: Get Items From O
Fullstack Starter
127.0.0.1:8080
Eleven Fifty
Apps
Video Courses
Banking
Fantasy Sports
Gaming
Work
Social
Developers, APIs, RSS
GitBook
Eleven Fifty SIS
API Development Too
4 GET and display json /test/one Display data
5 POST Data /test/seven Post
Posted data:
At:
6 POST Sign Up /authtest/createuser Username Password Submit See Console
7 Function: Get Session Token Print Token See Console
8 Authenticated Request /authtest/getall GET See Console
9 Authenticated POST Request to /authtest/create Enter Data POST See Console
10 GET Single Item /authtest/id Post ID # Get Single Item
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Audits
top
Filter
Default levels
Group similar
1 hidden
03-auth-test.js:20
(2) [{…}, {…}]
0: {id: 10, authtestdata: "A strange game. The only winning move is not to play.", owner: 13, createdAt: "2018-03-27T19:35:31.634Z", updatedAt: "2018-03-27T19:35:31.634Z"}
1: {id: 11, authtestdata: "How about a nice game of chess?", owner: 13, createdAt: "2018-03-27T19:35:44.607Z", updatedAt: "2018-03-27T19:35:44.607Z"}
length: 2
__proto__: Array(0)

Creating an Item for a User

05 - Creating an Item for a User

In this module, we'll make an Authenticated `POST` request to an authenticated `POST` route and persist some simple data in the database.

Code

Let's start by adding the following code to `03-auth-test.js`:

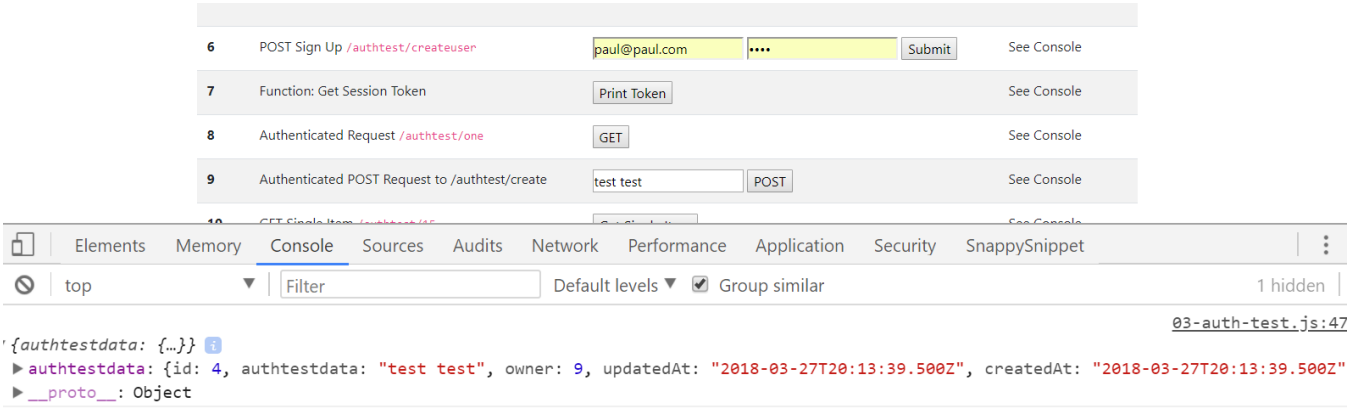
```
/*************************************
 * FETCH/POST to Auth/Create
************************************/
function postToAuthRouteCreate() {
    const fetch_url = `http://localhost:3000/authtest/create`
    const accessToken = localStorage.getItem('SessionToken')

    let authTestDataInput = document.getElementById('authTestData').value; //1

    let authInputData = { authtestdata: { item: authTestDataInput } }; //2

    const response = fetch(fetch_url, {
        method: 'POST', //3
        headers: {
            'Content-Type': 'application/json',
            'Authorization': accessToken
        },
        body: JSON.stringify(authInputData)  //4
    })
        .then(response => {
            return response.json();
        })
        .then(data => {
            console.log(data)
        })
}
```

Analysis

1. We will be using an input field in the DOM for this exercise, so we will grab whatever string that a user passes into that field.
2. We package that value up into an object. Notice that this object is similar in syntax to what we did with Postman when posting data.
3. Note that we are identifying this method as a POST request. If you are struggling with request problems, it's a good idea to take a look at your HTTP verb and make sure that you are using the right one. Since the server endpoint requires a POST request, we have to send the data as a POST request. GET would not work because we did not write our server endpoint as a GET request.
4. We package up the object as a JSON string and add it to the body of our request. The `JSON.stringify()` method will take a JS object and convert it into JSON.

Test

1. Make sure your client and server are both running.
2. Go to step 10 (Authenticated POST Request to /authtest/create) and enter something into the input field.
3. Press send, then check the console. You should see something similar to following response:



4. Note that the response will hold whatever value that you added to the input field.
5. Just for practice and orientation, it's also a good idea to go and see how this data has saved to Postgres. Refresh the `authtestdata` table and redo the query.

Task

For the next few modules you'll need an `id` number of the item you've just created. For instance, in the screenshot above, our `id` number is '4'. Yours will probably be '1' or '2'. Just make sure that you remember that number value.

Get One Item

06 - Get one item

In this module, we'll write a GET request that grabs a single item from the database for a specific user.

CODE

Add the following method to the `03-auth-test.js` file:

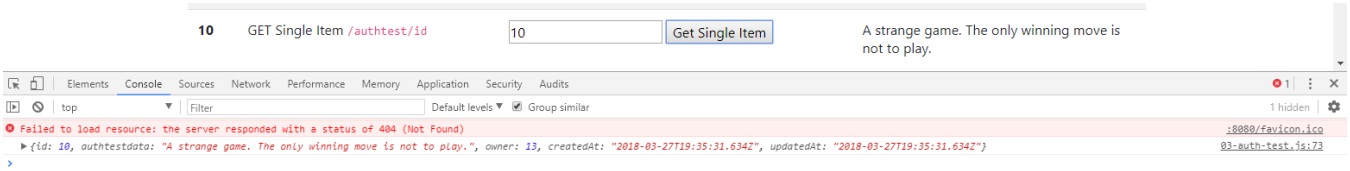
```
/*************************************
 * GET ITEM BY USER
*************************************/
function getOneByUser() {
    let postIdNumber = document.getElementById("getNumber").value; //1

    const fetch_url = `http://localhost:3000/authtest/${postIdNumber}` //2
    const accessToken = localStorage.getItem('SessionToken')

    const response = fetch(fetch_url, {
        method: 'GET',
        headers: {
            'Content-Type': 'application/json',
            'Authorization': accessToken
        }
    })
        .then(response => {
            return response.json();
        })
        .then(function (response) {
            console.log(response);
            var myItem = document.getElementById('getItemValue'); //3
            myItem.innerHTML = response.authtestdata; //4
        })
}
```

Analysis

1. We get the `postIdNumber` provided in the `getNumber` field. Because we are making an authenticated request, this `id` has to exist in the database, as well as match the `user.id` from the database for the user that you are using currently logged in as.
2. We pass the `postIdNumber` into the url with a template literal.
3. We target an element called `getItemValue`. It's a `label` tag.
4. We set the value of the label to the value of `response.authtestdata`. This means that the data will be populated in the label when the DOM loads.

Test

1. Go ahead and run the app.
2. Click the `Get Single Item` button after putting in the `id` from the previous module.
3. You should see the following:



4. Notice that we have pulled the single item from the server. In this case, the item has the id of `10`.
5. Not only did we print to the console, we added the data pulled from the server to the DOM in the `<label>` element on step 11 (GET Single Item).

Update an Item-3

07 - Update an Item

In this module, we'll update the item that is returned from the GET request and displayed in the input field.

CODE

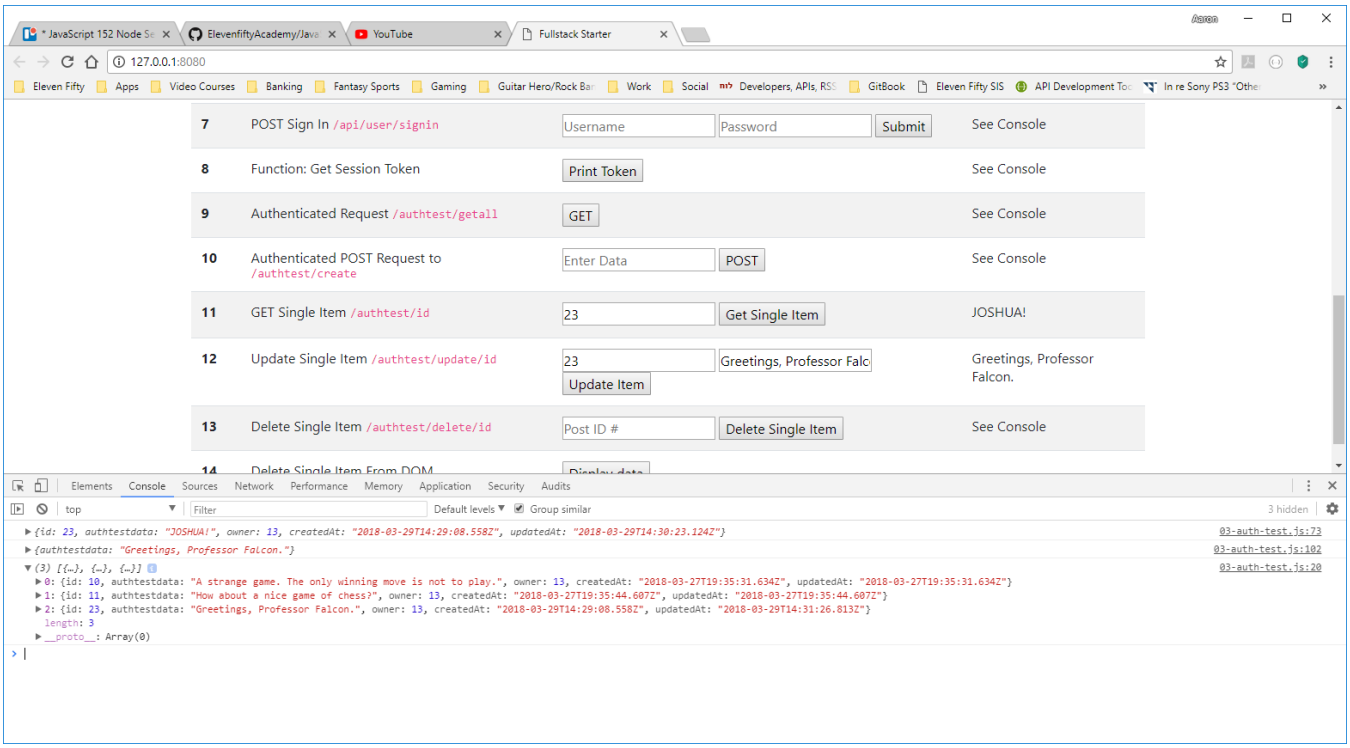
Add the following code to your `03-auth-test.js` file.

```
/*************************************
 * PUT to authtest/update/:id
**************************************/
function updateItem() {
    let postIdNumber = document.getElementById("updateNumber").value;
    let authTestDataInput = document.getElementById('updateValue').value; //1

    const fetch_url = `http://localhost:3000/authtest/update/${postIdNumber}` //2
    const accessToken = localStorage.getItem('SessionToken')

    let authInputData = { authtestdata: { item: authTestDataInput } }; //3
    const response = fetch(fetch_url, {
        method: 'PUT', //4
        headers: {
            'Content-Type': 'application/json',
            'Authorization': accessToken
        },
        body: JSON.stringify(authInputData) //5
    })
        .then(response => {
            return response.json();
        })
        .then(data => {
            console.log(data) //6
            var myItem = document.getElementById('newItemValue') //7
            myItem.innerHTML = data.authtestdata; //7
            fetchAllFromAuthRoute(); //8
        })
}
```

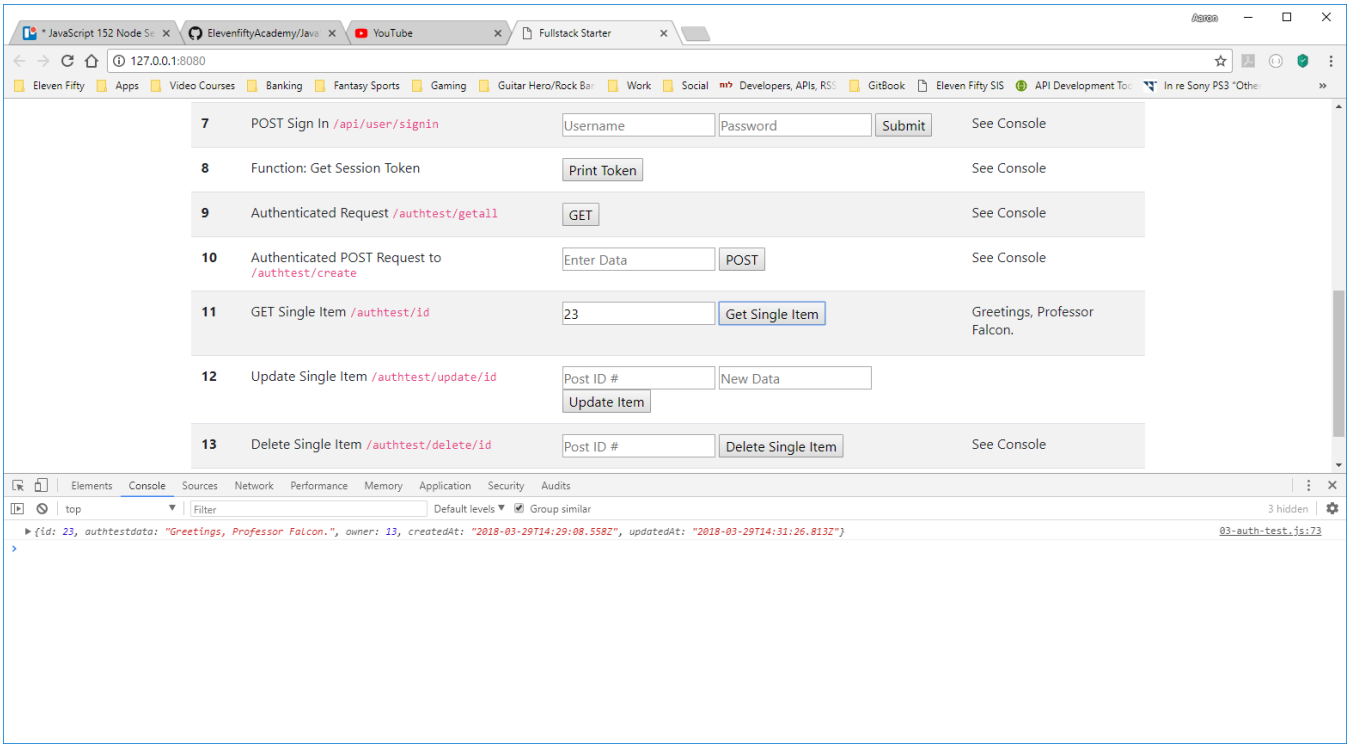
1. We get the value of the input provided from the user for both the `updateNumber` and `updateValue` fields and assign each to a variable.
2. Like before, we pass in the input from the user to the url with a template literal.
3. We create an object that packages up our request. We capture the value of `authTestDataInput` and store it in the variable `authInputData` variable.
4. We are doing an update method, so this will be a `PUT` request.
5. Just like we did in past `POST` methods, we use the `stringify` method to convert the object to a JSON object.
6. We print the response to our fetch to the console.
7. We make a reference to the `<label>` in step 12 (Update Single Item), then set its value to the data we put in the database.
8. We run the `getall` function again and print the new contents of the database to the console.

TEST

1. Make sure that the server and client are running.
2. Press the `GET Single Item` button on Step 11. That label should populate.
3. Change the text in the input field while keeping the same ID# in the input field of Step 12 so that it is something different than what you had.
4. Press the `Update Item` button.
5. You should see the following response in the console. A label will appear in Step 11 containing the new data, and the current contents of the database should also print to the console after the update runs:



6. Go ahead and refresh the page. You don't need to turn off the Client or Server.
7. Press the `GET Single Item` button on Step 11. The label should populate again.
8. You should see that the newly updated data in the label and console message:



9. We might also suggest that you check in Postgres to be sure that you have successfully added the data to the database.

Getting Fancy

While we're here, let's mess around about with the stuff we're getting from the database. Go to your `index.html` file and look for Step 12. We're going to add a little bit to one of the input fields:

```
                <tr>
                    <th scope="row">12</th>
                    <td>Update Single Item
                        <code>/authtest/update/id</code>
                    </td>
                    <td>
                        <input type="text" id="updateNumber" placeholder="Post ID #" onkeyup="showCurrentData(this)"/> <----- ADD T
HIS
                        <input type="text" id="updateValue" placeholder="New Data"/>
                        <button onclick="updateItem();">Update Item</button>
                    </td>
                    <td>
                        <label id="newItemValue"></label>
                    </td>
                </tr>
```

You've used listeners in the past, but this time instead of adding one in JavaScript, we're using the HTML attribute `onkeyup`. This listener looks for a key on the keyboard being released; in other words, after you press the key down, it waits until a key is released then does whatever you tell it to.
We're adding the listener to the `updateNumber` input field, so anytime a key is released while that field is focus (i.e. we're typing something in the field), the function attached to the listener will run. `this` is passed as a parameter into our function. Here, `this` refers to the input field itself, which will allow us to easily access the value of the field in out function.
If you look at the button, you can see another one of these listener attributes `onclick`. That one allows us to make the button do something without having to make it a `submit` type. In fact, all of the buttons on this page have that listener attached.

showCurrentData()

The goal of this listener is to automatically search our database for an `id` that matches what we've entered into the `updateNumber` field, then populate the `updateValue` filed with the data that is returned. Add this code to the bottom of your `03-auth-test.js` file:

```
function showCurrentData(e) { //1
    const fetch_url = `http://localhost:3000/authtest/${e.value}` //2
    const accessToken = localStorage.getItem('SessionToken')

    fetch(fetch_url, {
        method: 'GET',
        headers: {
            'Content-Type': 'application/json',
            'Authorization': accessToken
        }
    })
        .then(response => {
            return response.json();
        })
        .then(function (response) {
            console.log(response);
            var myItem = document.getElementById('updateValue'); //3
            if (!response) return; //4
            else myItem.value = response.authtestdata; //5
        })
}
```

Analysis

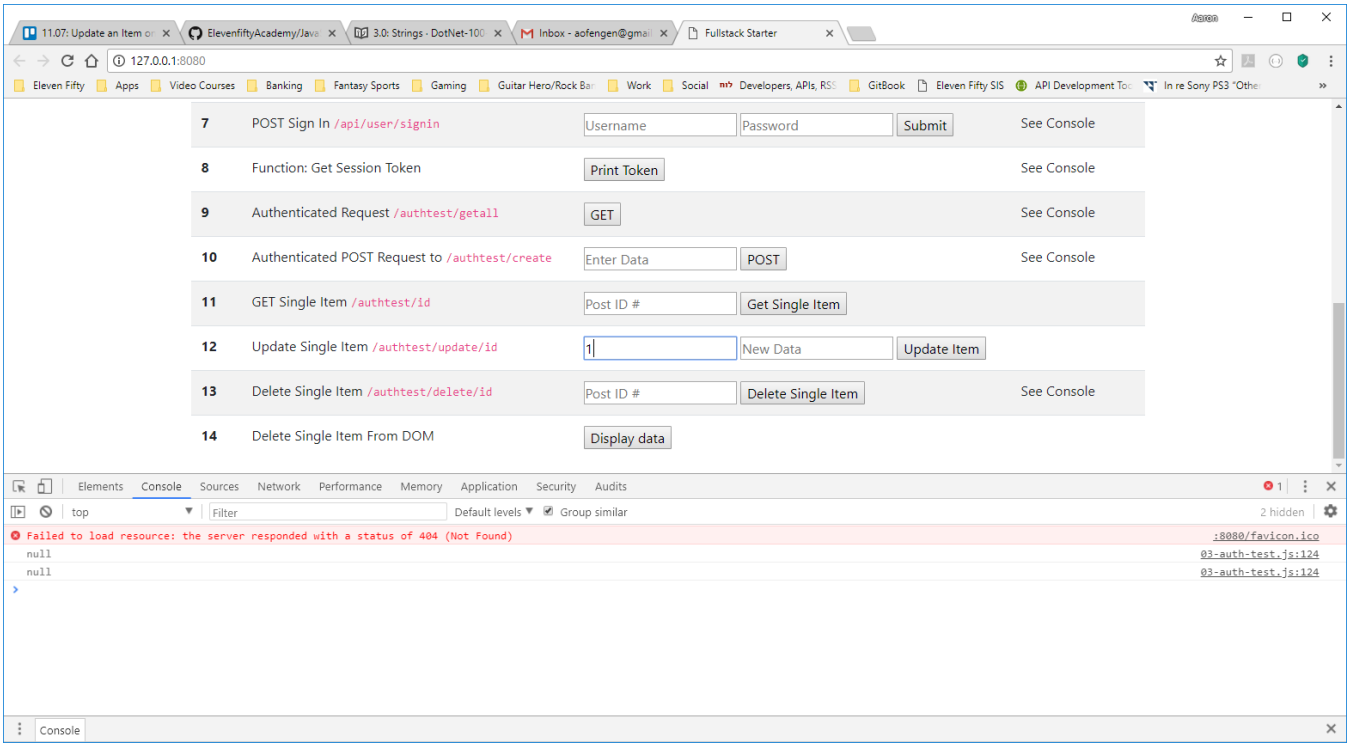
1. `e` is the default variable name for an Event Listener. Here, `e` represents the input field `updateNumber`, which was passed as a parameter using `this` on the HTML page.
2. We pass the value of the input field supplied by the user directly into the URL with a template literal. Because `e` is already defined as the input field, we don't need to use a function to get another reference to it.
3. We call the DOM element we want to modify and set it to a variable to be accessed later.
4. If no item in the database matches the `id` we've supplied, the response comes back undefined. A blank return statement tells the program not to return anything and just to move on. Remember that not only does the `id` have to match what's in the database, but `user.id` also has to match the `owner` property, signifying that the current user is the one who entered it.

5. We could use `innerHTML` to set the value, but that method doesn't work with `<input>` elements. Instead, we use `value` to insert our data into the field.

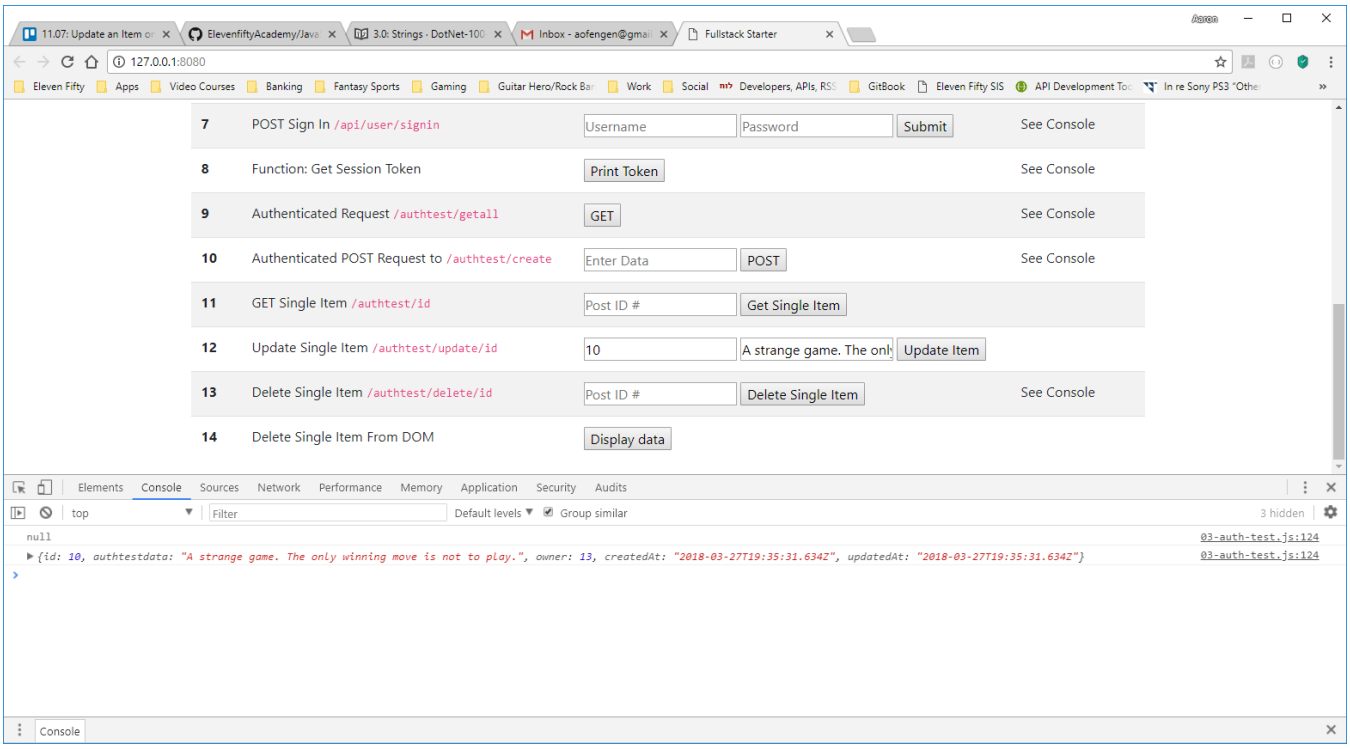
Again, this function will not run until a key is lifted after being pressed.

Testing

1. Your client and server should still be running; if not, restart them.
2. Refresh your browser, then type a number into the first field in step 12.
3. If nothing in the database matches the `id` you've entered, you should see something like this:



4. If a match is found, you should see something like this:



5. You can then change the text however you'd like. Click `Update Item` and refresh your database to make sure it worked.

This process allows you to see what's in the database before you replace it. Most users, if not all, won't have direct access to the database, so this is one way that can show them what's already saved for a particular item.

Deleting an Item

08 - Deleting an Item

In this module, we'll write a method to delete a single item in the database.

CODE

Add the following method to the `03-auth-test.js` file:

```
function deleteItem() {
    let postIdNumber = document.getElementById("deleteNumber").value;

    const fetch_url = `http://localhost:3000/authtest/delete/${postIdNumber}` //1
    const accessToken = localStorage.getItem('SessionToken')

    const response = fetch(fetch_url, {
        method: 'DELETE', //2
        headers: {
            'Content-Type': 'application/json',
            'Authorization': accessToken
        }
    })
        .then(response => { //3
            console.log(response);
            fetchAllFromAuthRoute()
        })
}
```

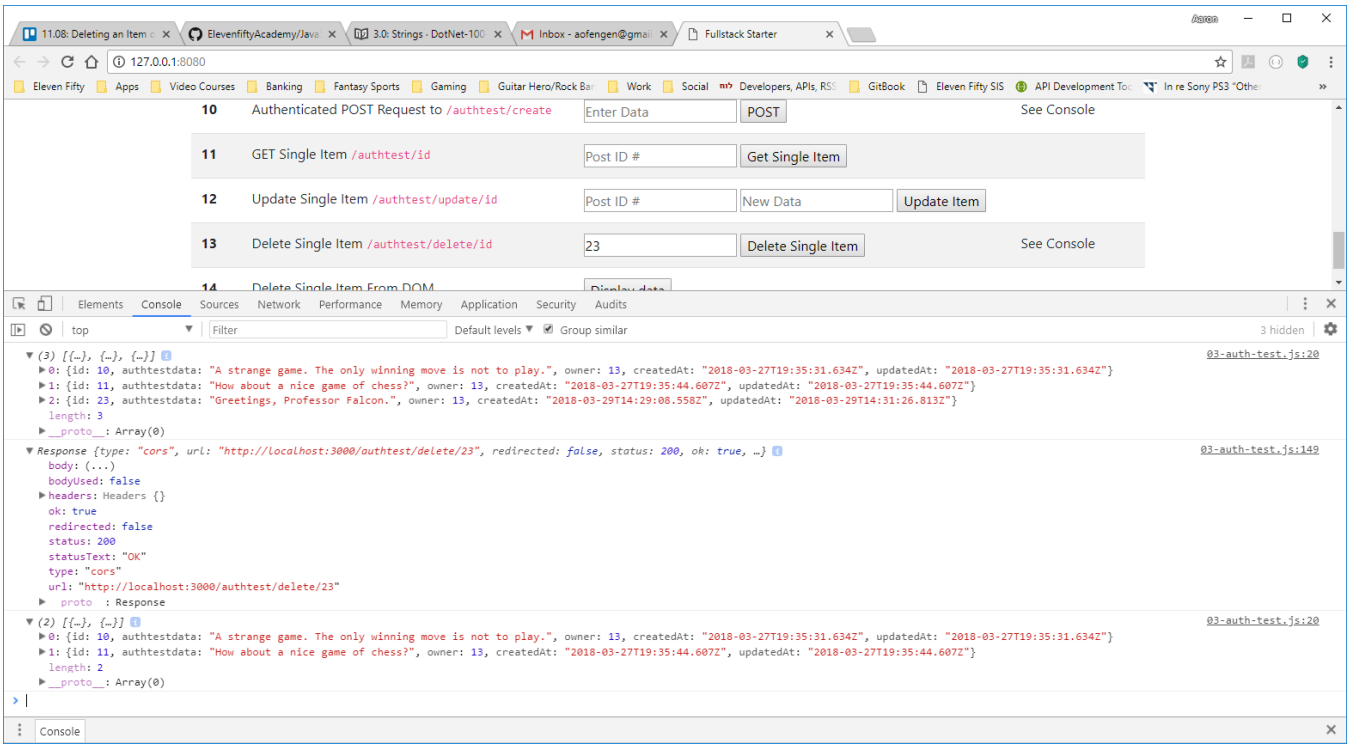
Analysis

This method is simple so far. Let's do three simple explanations.

1. Again we get the `id` number submitted by the user and pass it into the url via a template literal.
2. Our HTTP verb is DELETE in this case, so we use the `DELETE` method.
    1. We print the response to the console and also run the `fetchAllFromAuthRoute` function again, which will print all remaining items for our user to the console.

Test

1. Run the client and server.
2. Run Step 9 (Authenticated Request) first to print the current contents of the database to the console. This will show you the `id` value for each item.
3. Enter the `id` number for the entry you want to delete into the field in step 13, then click `Delete Single Item`.
4. You should see the following response:



5. You can see both the response to the `DELETE` request and the contents of the database after the item was deleted. Refresh your database to make sure that it worked properly.

Deleting with a Custom Event

09 - Deleting with a Custom Event

The last step we have is to display a list of all the items for a given user. However, we're going to add a custom click listener to each item so that when item is clicked, it immediately deletes that item from the database. We'll do the display function first, then the delete.

Display All Items for a User

Put this at the bottom of your `03-auth-test.js` file. We'll cover what's happening here in a moment:

```
function fetchFromOneDisplayData() {
    const url = 'http://localhost:3000/authtest/getall';
    const accessToken = localStorage.getItem('SessionToken')

    fetch(url, {
        method: 'GET',
        headers: new Headers({
            'Content-Type': 'application/json',
            'Authorization': accessToken
        })
    }).then(
        function (response) {
            return response.json()
        })
        .catch(
            function (error) {
                console.error('Error:', error)
            })
        .then(
            function (response) {
                let text = '';
                var myList = document.querySelector('ul#fourteen'); //1
                while (myList.firstChild) { //2
                    myList.removeChild(myList.firstChild)
                }

                console.log(response);
                for (r of response) { //3
                    var listItem = document.createElement('li'); //4
                    var textData = r.id + ' ' + r.authtestdata; //5
                    listItem.innerHTML = textData;
                    listItem.setAttribute('id', r.id); //6
                    myList.appendChild(listItem); //7
                    myList.addEventListener('click', removeItem);  //8
                }
            })
}
```

Analysis

1. This is a little different way of making a reference to a DOM element. We're aiming for a `<ul>` element with an `id` of `fourteen` (the `#` signals the program to look for an `id` rather than a `class`).
2. This should look familiar to you. This is the same way we cleared out the `<section>` elements in the NYT and YouTube API mini-apps.
3. We use a `for of` loop to iterate through the values of each `key: value` object pair.
4. Given that we're working with a `<ul>` element, each loop will create a different `<li>`.
5. We create a string with the `id` and `authtestdata` properties, then put that string into the `<li>` element.
6. We add the `id` property of each object as an id for each `<li>`. This will allow us to call them individually later.
7. The `<li>` child element is added to the end of the `<ul>` parent element.
8. We add our custom listener to run whenever an `<li>` is clicked.

We need to create the `removeItem` function before we test this part, or the app will break.

removeItem()

This function will delete an item from the `<ul>` element. It will also have the ability to send a `DELETE` request, but we're going to hold off on that part for a second. Add this function at the bottom of the `03-auth-test.js` file:

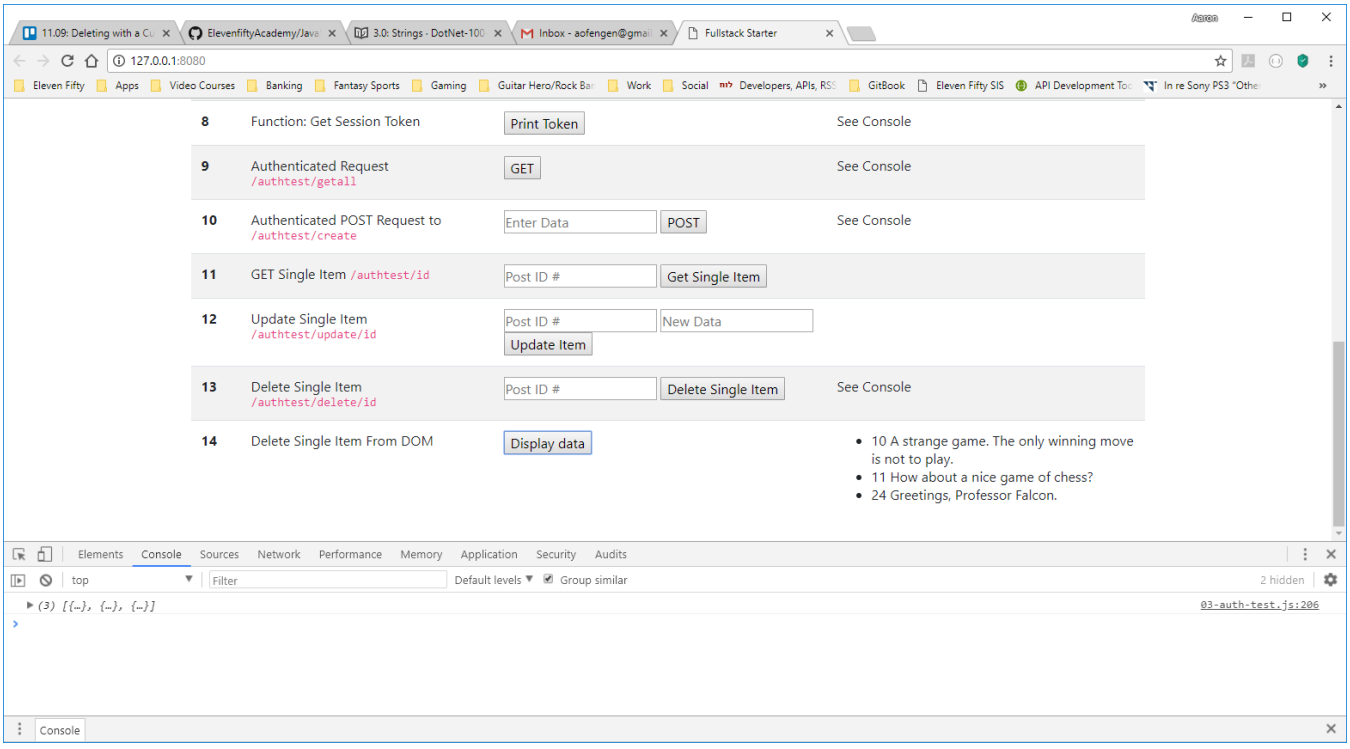
```
function removeItem(e) {
    console.log(e); //1
    var target = e.target; //2
    if (target.tagName !== 'LI') return; //3
    else target.parentNode.removeChild(target); //4

    let x = target.getAttribute("id") //5
    //deleteItemById(x); //6
    console.log("The id number for this item is " + x);
}
```

Analysis

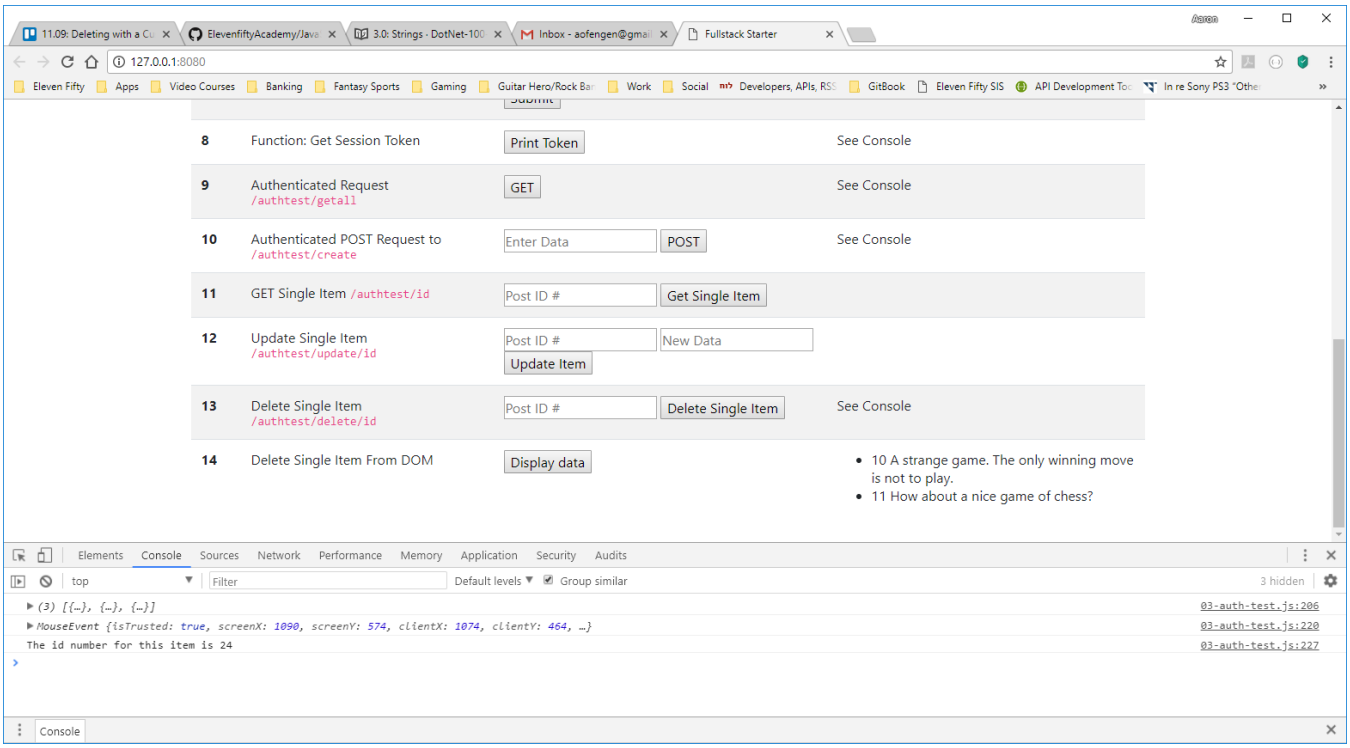
1. Print `e` to the console to check which item we're clicking on.
2. `target` is a nested object within `e`. This places that object inside its own variable.
3. If the item we're clicking on isn't an `<li>` element, the empty `return` statement exits the conditional.
4. We remove the `<li>` child from the `<ul>` parent.
5. Earlier we set an `id` for the `<li>`. Now we get it back so we can pass it to the `DELETE` request.
6. This will become our `DELETE` request. In order for us to test what we have so far, we'll just print `x` to the console.

Testing

1. If they aren't already running, start your client and server.
2. Refresh your browser, then click on `Display data` in step 14 (Delete Single Item From DOM).
3. You should see the items from the `authtestdata` table with the matching `owner` value for our current user token:



4. Click on one of the items. It should disappear from the list and print something like the following to the console:



Now that we can remove the element from its parent, we need to send a `DELETE` request to remove the item from the database. One last step, and we're done!

deleteItemById()

Since this is a `DELETE` request, put this function between the `deleteItem()` and `fetchFromOneDisplayData()` functions:

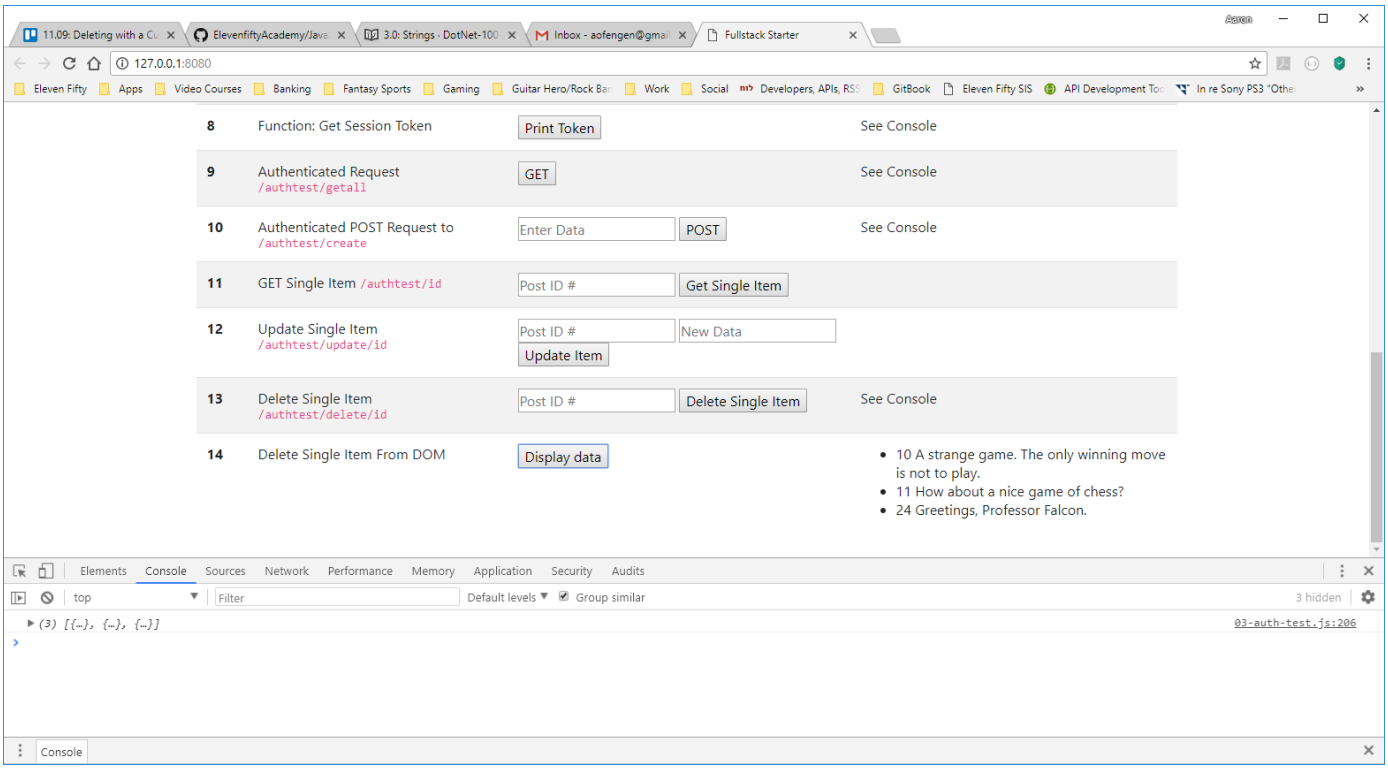
```
function deleteItemById(paramNum) { //1
    const fetch_url = `http://localhost:3000/authtest/delete/${paramNum}`
    const accessToken = localStorage.getItem('SessionToken')

    const response = fetch(fetch_url, {
        method: 'DELETE',
        headers: {
            'Content-Type': 'application/json',
            'Authorization': accessToken
        }
    })
        .then(response => {
            console.log(response);//2
            fetchAllFromAuthRoute();//3
        })
}
```

Analysis

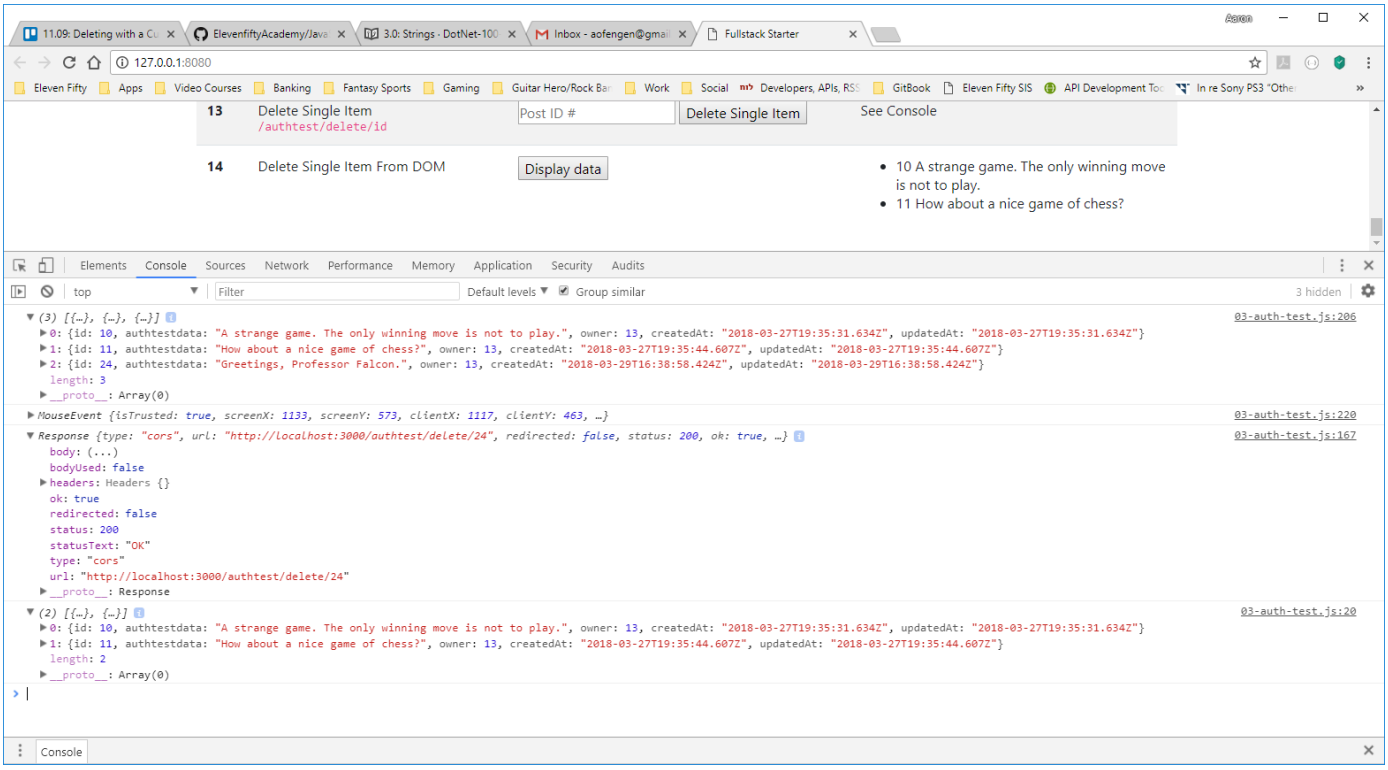
Before we go further, go to the `removeItem()` function. Comment out the last line (the `console.log`) and uncomment out the line before it, which triggers this function. This function is nearly identical to the previous delete method, so it should look familiar.

1. The `id` of the `<li>` is passed into this function as a parameter, which is then added to the url via the template literal.
2. Print the response to the console to verify the delete worked.
3. Run the `getall` function again to print the remaining items in the database to the console.

Now for the final test. Follow the instructions from the Testing section above, but this time, you should see something like this:
Pre-delete



Post-delete



Wrap Up

And that's it! That's everything you need to build your own CRUD app, both client side and server side. If there's anything that you find unclear, please ask us so that we can help, but try on your own first. The next step is a new challenge, so when you're ready, move on to the next chapter.