Name: William Reed

Student ID: Z23564142

GitHub Username: willmreed14

GitHub Repo: https://github.com/cop4808-spring-2023-fullstack-web/hw7-ejs-mongodb-willmreed14

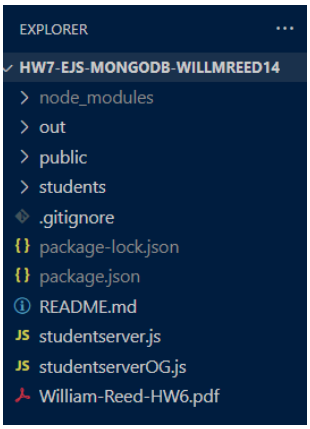
Date: 3/15/2023

Time: 9:00 P.M.

Homework 7

Objective 0: Preparation

The first step in beginning this assignment was preparing my repository with the necessary files to start. I did this by first copying my Homework 6 repository and placing the copy in my Homework 7 repository.



Objective 1: Convert Front-End to EJS

The next step was to introduce EJS to my project. To do this, I began by adding the following lines to the studentserver.js file:

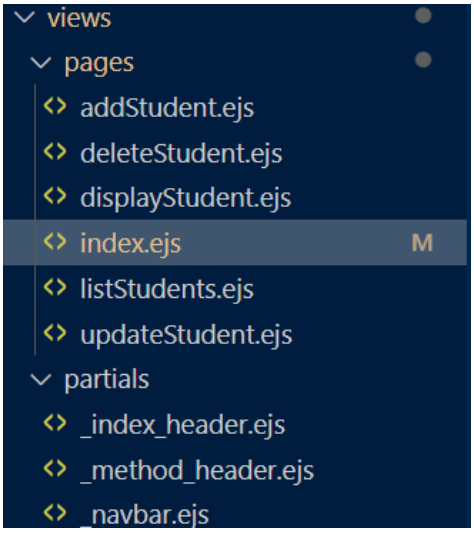
Require the ejs node module.

```
const ejs = require('ejs');
```

Set the server's view engine to ejs.

```
app.set("view engine","ejs");
```

Next, I had to create a folder in my repo directory called "views" to store the ejs templates. In the views folder, I created two sub-folders, pages and partials. Pages stores the full ejs pages for front-end display, while partials stores ejs templates for use in the dynamic html pages.



Now that my folder were set up, I could start implementing ejs. To start, I copied every html file from "public" and pasted the content into a corresponding ejs file in /views/pages. Once that was done, I deleted all static html files, as they would no longer be used in this project.

The next step was to create my partial templates. I used three templates in this project: index_header, method_header, and navbar. To create these templates, I copied the relevant portion of each from the full html into its own ejs file in /views/partials.

Index Header:

views > partials > _index_header.ejs > link

```
1   <meta charset="utf-8" />
2   <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no" />
3   <meta name="description" content="" />
4   <meta name="author" content="" />
5   <title>Student Server</title>
6   <!-- Favicon-->
7   <link rel="icon" type="image/x-icon" href="assets/favicon.ico" />
8   <!-- Core theme CSS (includes Bootstrap)-->
9   <link href="css/styles.css" rel="stylesheet" />
```

Method Header:

views > partials > _method_header.ejs > link

```
    <!-- Including jQuery-->
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <!-- Favicon-->
    <link rel="icon" type="image/x-icon" href="assets/favicon.ico" />
    <!-- Core theme CSS (includes Bootstrap)-->
    <link href="css/styles.css" rel="stylesheet" />
    <!-- Include Bootstrap for styling -->
    <link rel="stylesheet"
          href=
    "https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">
    <!-- Include the Bootstrap Table CSS
    for the table -->
    <link rel="stylesheet"
          href=
    "https://unpkg.com/bootstrap-table@1.16.0/dist/bootstrap-table.min.css">
```

Navbar:

views > partials > _navbar.ejs > div.container > div#navbarSupportedContent.collapse.navbar-collapse > ul.navbar-nav.me-auto.mb-2.mb-lg-0 >

```
<div class="container">
    <a class="navbar-brand" href="/">HW7</a>
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent
    <div class="collapse navbar-collapse" id="navbarSupportedContent">
      <ul class="navbar-nav me-auto mb-2 mb-lg-0">
        <li class="nav-item">
          <a class="nav-link" href="/">Home</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="/pages/addStudent">Add</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="/pages/updateStudent">Update</a>
        </li>
```

Finally, I needed to render the ejs pages since there were no longer any static HTML pages for the app to use as a front end. To do this, I used a GET request in the studentserver.js file for each ejs page.

```
// *** Page Display Route Handling ***
// I need to GET the ejs pages and display them on the front end.

// Root: Home page (index.ejs)
app.get("/", function (req, res){
    res.render('pages/index');
  });
```

Each GET request for an ejs page works in the same way. I provided the request with an endpoint to request from, in this case, the root directory. I then fired a function with a request and a response, and finally told the server to render the requested page and send it as the response. I repeated this process for all ejs pages.

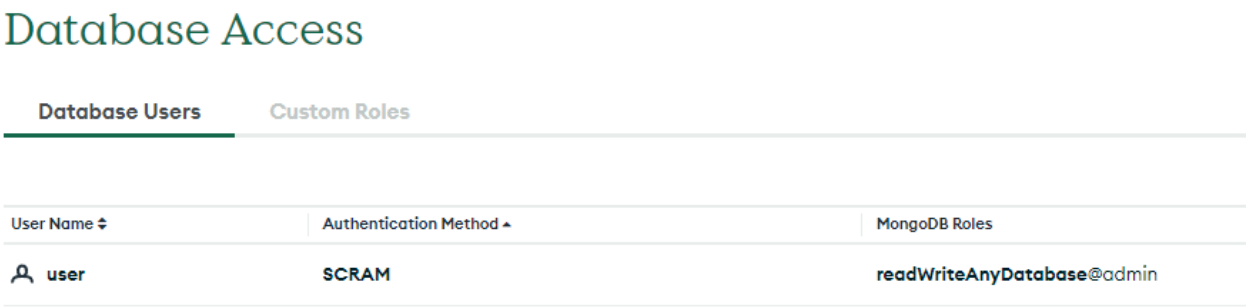
At this point, ejs was now fully functional in my student server app.

Objective 2: Convert Back-End to MongoDB

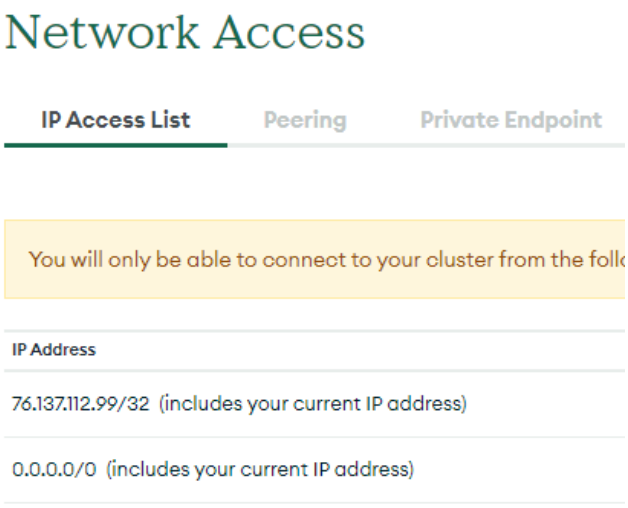
I began this objective by creating an account on MongoDB Atlas, the cloud-service version of MongoDB.

Next, I created an online Atlas cluster called “Will-Reed-Cluster-0” which would be used to connect to my Mongo database.

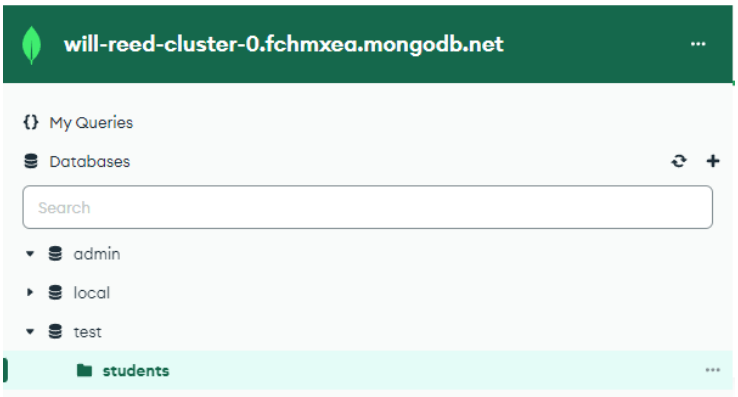
In the setup for this cluster, I configured the username, “user” and password, also “user”.



I also made sure that, for the purpose of this assignment, the database could be accessed by any IP address.

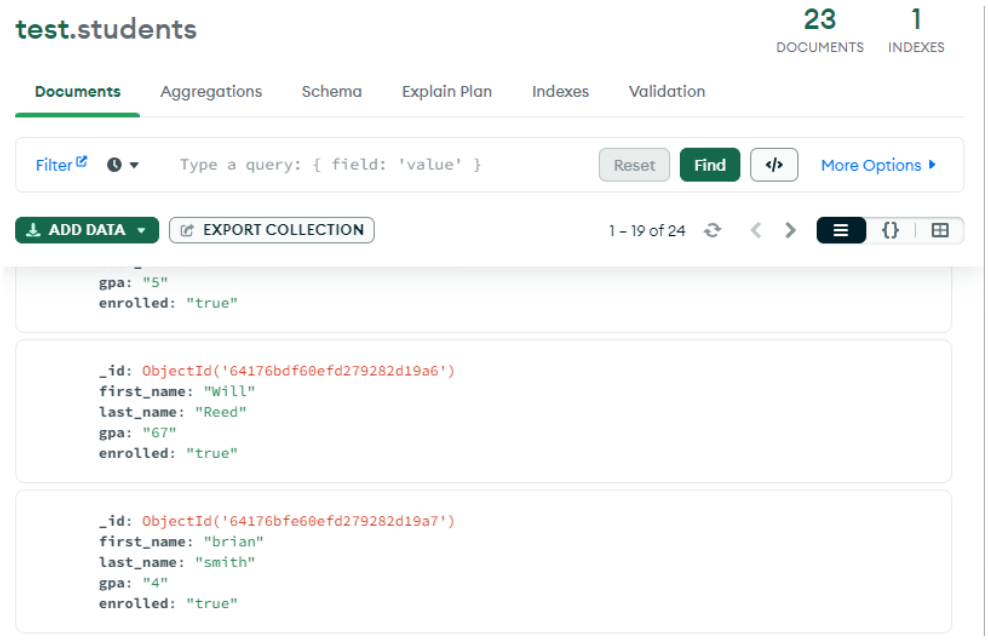


Once Atlas was fully setup to handle my database, I installed MongoDB compass on my desktop. This enabled me to view my database information and the documents stored in it on my desktop, without having to go through a web browser.



As you can see, I have MongoDB compass connected to my Atlas cluster, and the databases are listed below the cluster URI.

Looking further in Compass, I can also find the documents (students) in the students collection of my database, which is simply called “test”.



Now that MongoDB was fully setup outside of the code, it was time to get to the JavaScript.

First and foremost, I made sure to require the MongoDB npm package in studentserver.js.

```
// include ObjectId and mongoDB
const { ObjectId } = require('mongodb');
```

Next, I used npm to install the mongodb package.

```
wcrv1@Polo_PC MINGW64 ~/Documents/GitHub/hw7-ejs-mongodb-willmreed14 (main)
$ npm install mongodb
```

Now that MongoDB was ready to be used in my code, I needed to connect the server to the database. I did this by creating a separate file called ‘db.js’, which would be used to handle most of the tasks for connecting to the database.

```
module.exports = {
    connectToDb: (cb) => { // initially connect to a database.
        /* when we call connectToDb, we have to pass it a function (cb) as an argument,
        which is a function we want to run after connection is established. */
        MongoClient.connect(uri) //connect to local database Student-Server
            .then((client) => { // after connecting, we get access to 'client', client is an object we created by connecting
                dbConnection = client.db() // client.db is the connection returned, lets extract that into a variable
                return cb()
            })
            .catch(err => { // this occurs if it fails to connect to the MongoDB database
                console.log(err)
                return cb(err) // if err occurs, pass it to cb func as an argument
            })
    },
    // at this point we are either connected or threw an error

    getDb: () => dbConnection // return a value: the database connection
}
```

The db.js file works by establishing a function, connectToDb, that will be passed a callback function to execute when it is finished connecting to the database. The mongoClient connects to to the database via the uri, which is provided by MongoDB atlas. I stored the actual URI in a variable called uri, to make it easier to use.

```
let uri = 'mongodb+srv://user:user@will-reed-cluster-
0.fchmxea.mongodb.net/?retryWrites=true&w=majority'
```

Moving back to the studentserver.js file, I used the code from db.js to connect to the database. I configured it so that the server would not begin running on port 5678 until it was successfully connected to the database. I did this because if the server began running but the database failed to connect, the user could try to submit requests that would have no way of being handled.

```
let db; // this will store the connection once we connect
connectToDb((err) => { // call connectToDb (from db.js) and pass a callback
function as the argument
    // if the connection is a success, err argument will be null since we dont
pass an err on a success
    // if the connection fails, the err will have a value.
    if (!err){
        // start listening for requests on port 5678 AFTER we know connection was
a success.
```

```
        app.listen(5678, () => {
            console.log('app is listening on port 5678');
        });
        db = getDb(); // give me the database connection object so I can use it.
    }
});
```

Finally, my server is connected to the MongoDB database.

At this point, it was time to start implementing the data access for requests. Each request method followed a similar structure: access the necessary file or directory, perform the desired action, and respond with the necessary information.

Show implementation/code of each of the following endpoints (25 points - 5 points each)

GET: Display all students.

The implementation for getting all students starts with an app.get request in the /students directory, which in this case, is the students collection of my database.

```
app.get('/students', (req, res) => {
```

I created an empty array, called students, which would store the information of each student we were going to “get”.

```
let students = []
```

Next, I used the .find() method to retrieve the set of students from the database collection "students".

```
        .find() // returns a cursor (object that points to the items of query)
```

```
                // toArray: Fetch all docs that cursor points to and put them in
an array
                // forEach: Iterates each doc to process them individually
                // no parameter in find() means we are not filtering the query
```

I also sorted the results of .find() by last name, using .sort():

```
.sort({ last_name: 1 }) //sort the results alphabetically by last name
```

Next, I added each student retrieved to my array from before, called students.

```
.forEach(student => students.push(student)) // add each student found to the
array of students
```

Finally, I added an asynchronous function that would send the response once the array was filled with all of the students retrieved.

```
        // only starts when all students being queried have been fetched and
pushed to the array
        .then(() => {
            res.status(200).json(students) /* respond w/ status 200 (all good!)
                                              and json string version of students
array */
        })
```

I also added a catch case, just in case an error of some kind were to occur, the server would know how to handle it.

```
        .catch(() => { // if we could not fetch the students
            res.status(500).json({error: 'Could not fetch the documents'}) //
respond with an error
        })
})
```

GET: Display one student

This implementation works similar to the one for getting all students. Instead of sending the request to the students directory, the user will provide the record ID of a specific student, and this request will get that student. In the GET implementation, I first used an if statement to check if the record ID provided by the user was valid. If valid, the request will continue to run, and if not, it will throw an error. Next, the function accesses the students collection. Then, it uses the .findOne method to find the document with the specified record ID. Finally, when the student document is found, it sends the document as a response.

```
// GET a single student
app.get('/students/:id', (req,res) => {

    if(ObjectId.isValid(req.params.id)){ // check if the id requested is valid
(i.e. is it in the correct format)
        db.collection('students') // access the students collection
        .findOne({_id: new ObjectId(req.params.id)}) //find the document with the
specified ID
        .then(doc => { // once the proper document is found, send the response
            res.status(200).json(doc)
        })
        .catch(err => { // if the document cannot be found, send an error
            res.status(500).json({error: 'Could not fetch the document'})
        })
    } else {
        res.status(500).json({error: 'ID is not valid'})
    }
})
```

POST: Create a new student

The POST implementation accesses the /students directory. I then defined a variable, student, to store the body of the request; this means store the information about the student, which is provided by the user in the front-end. Next, the function uses the .insertOne() method to insert a new student into the students collection of my database, and uses the student variable to insert

the body information in the new student document. Finally, after all of this is complete, the function sends the result as the response, in this case it is the student's record ID.

```
// POST: Create a new student

app.post('/students', (req, res) => {
    const student = req.body

    db.collection('students')
        .insertOne(student)
        .then(result => {
            res.status(201).json(result)
        })
        .catch(err => {
            res.status(500).json({err: 'Could not create student'})
        })
})
```

DELETE: Delete a student

The DELETE implementation accesses the /students/:id directory. The user will provide the record ID of a specific student, and this request will delete that student. Similar to the GET request for a single student, this function will also verify that the record ID is valid first. Next, it will access the students collection in the Mongo database. Then it uses the .deleteOne() method with the student ID to delete that student record, or document, from the students collection. Finally, it sends the json result as the response, in this case, the ID of the student that was deleted.

```
// DELETE: Delete a student

app.delete('/students/:id', (req, res) => {

    if(ObjectId.isValid(req.params.id)){ // check if the id requested is valid
(i.e. is it in the correct format)
        db.collection('students') // access the students collection
```

```
        .deleteOne({_id: new ObjectId(req.params.id)}) //find the document with
the specified ID
        .then(result => { // once the proper document is found, send the response
            res.status(200).json(result)
        })
        .catch(err => { // if the document cannot be found, send an error
            res.status(500).json({error: 'Could not delete the student'})
        })
    } else {
        res.status(500).json({error: 'ID is not valid'})
    }
})
```

PUT: Update a student

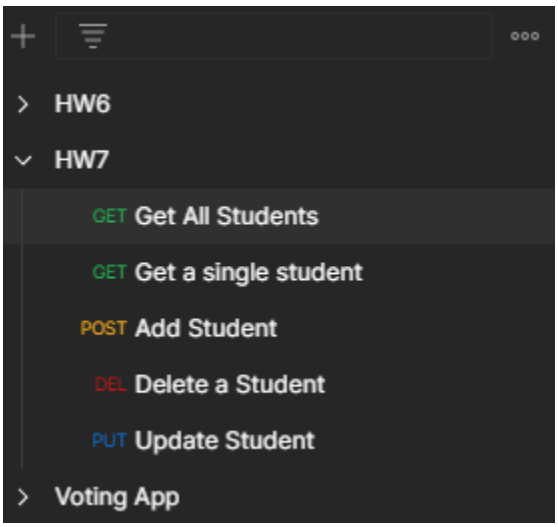
The PUT implementation accesses the /students/:id directory. The user will provide the record ID of a specific student, and the new information about the student (i.e. name, gpa, enrollment status) and this request will update that student. This function will also verify that the record ID is valid first. Next, it will access the students collection in the Mongo database. Then it uses the .updateOne() method with the student ID to update that student record, or document, in the students collection. Finally, it sends the json result as the response, in this case, the ID of the student that was updated.

```
// PUT: Update a student

app.put('/students/:id', (req, res) => {
    const updates = req.body

    if(ObjectId.isValid(req.params.id)){ // check if the id requested is valid
(i.e. is it in the correct format)
        db.collection('students') // access the students collection
        .updateOne({_id: new ObjectId(req.params.id)}, {$set: updates}) //find
the document with the specified ID
        .then(result => { // once the proper document is found, send the response
            res.status(200).json(result)
        })
        .catch(err => { // if the document cannot be found, send an error
```

```
            res.status(500).json({error: 'Could not updtae the student'})
        })
    } else {
        res.status(500).json({error: 'ID is not valid'})
    }
```

Test and show each of the routes(/methods/endpoints) defined using Postman (30 points - 5 points each

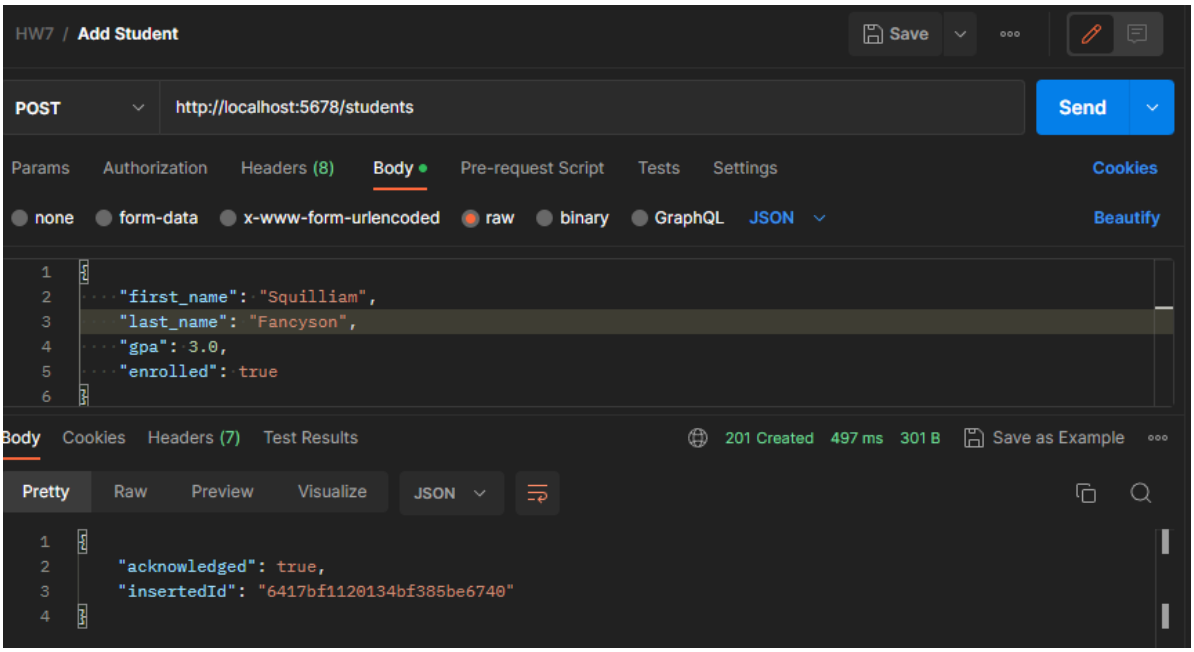
I started by storing each request in a collection called HW7 in Postman:



Next, I tested each request, starting with…

1. POST: Add Student

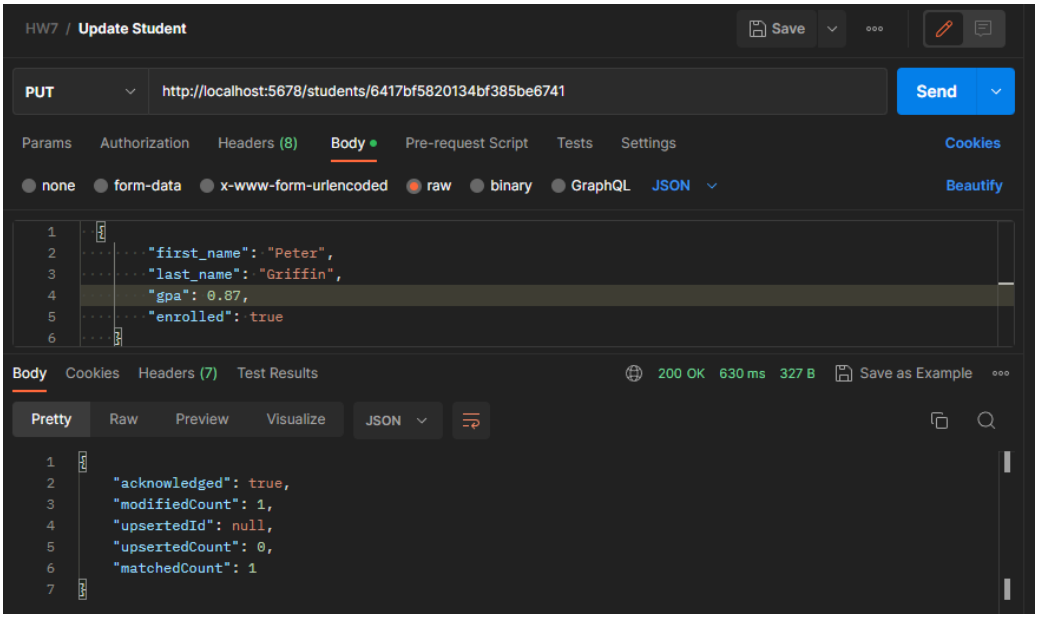
   I typed in the data about a new student, Squilliam, in the JSON body for this request. I made sure to enter the correct endpoint at the top, and pressed send.



   I repeated this process for two more students, Peter and Lois, so that I had more than one document to work with.

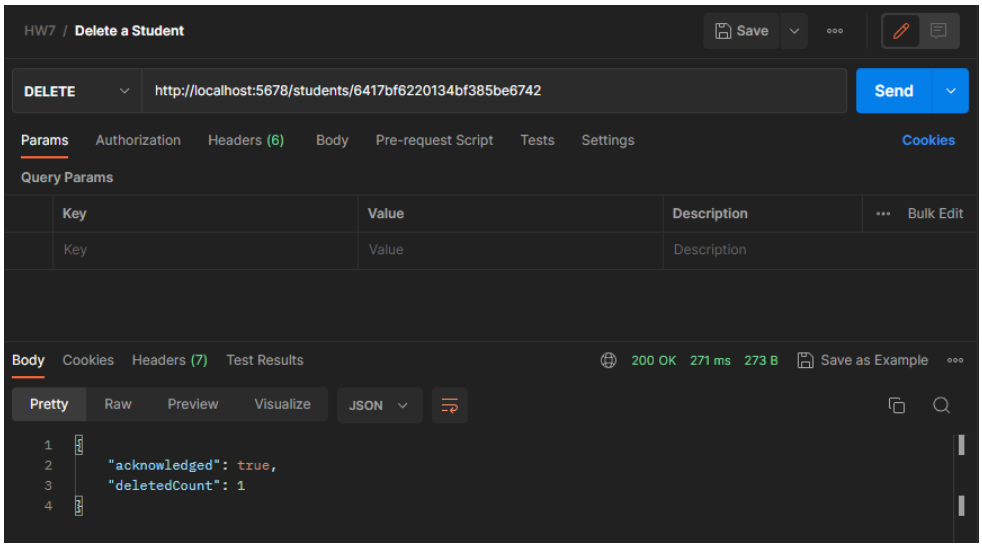
2. PUT: Update Student:

   I realized after creating the student Peter Griffin, that his GPA was far too high. So to update this, I used the PUT request, complete with his new GPA and the rest of the information about him. I used his record ID, 6417bf5820134bf385be6741, as the endpoint, so that the request knew which student I wanted to update.



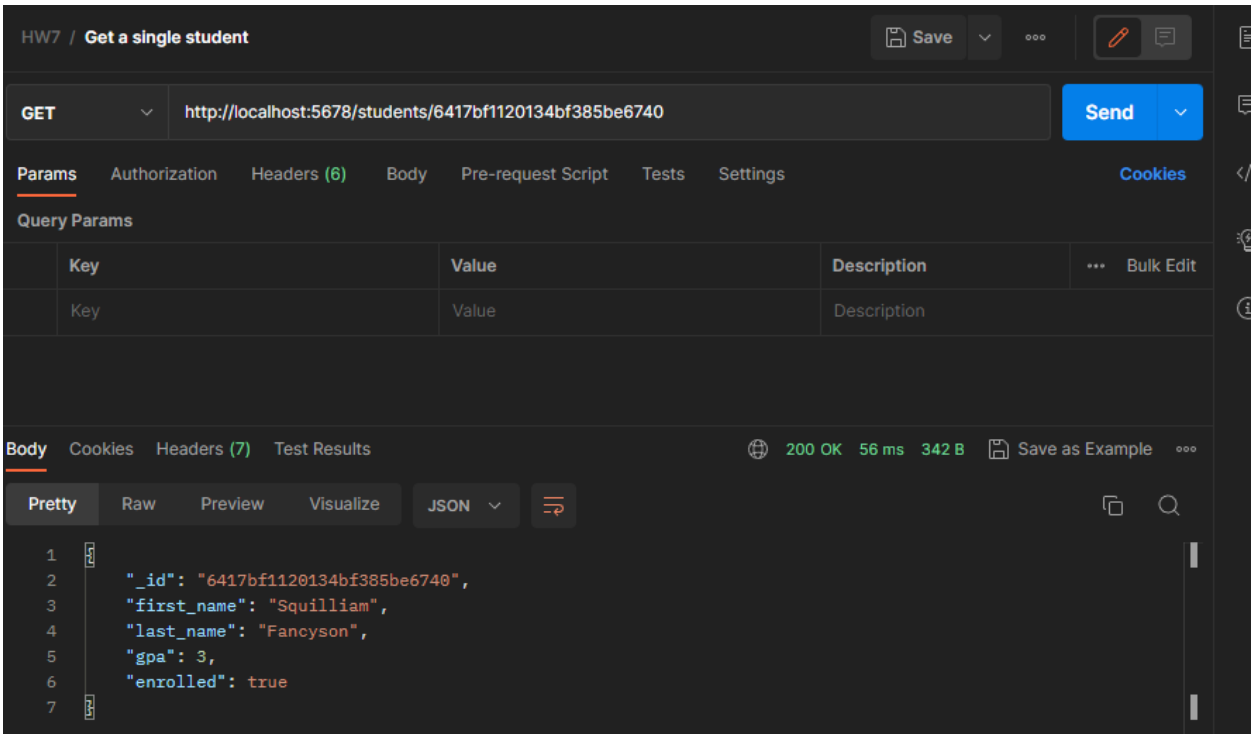
3. DELETE: Delete a Student:

   I realized too late that Lois transferred to a different school, so I needed to delete her from the students collection. I used the DELETE request, with her record ID as the endpoint. Goodbye, Lois.



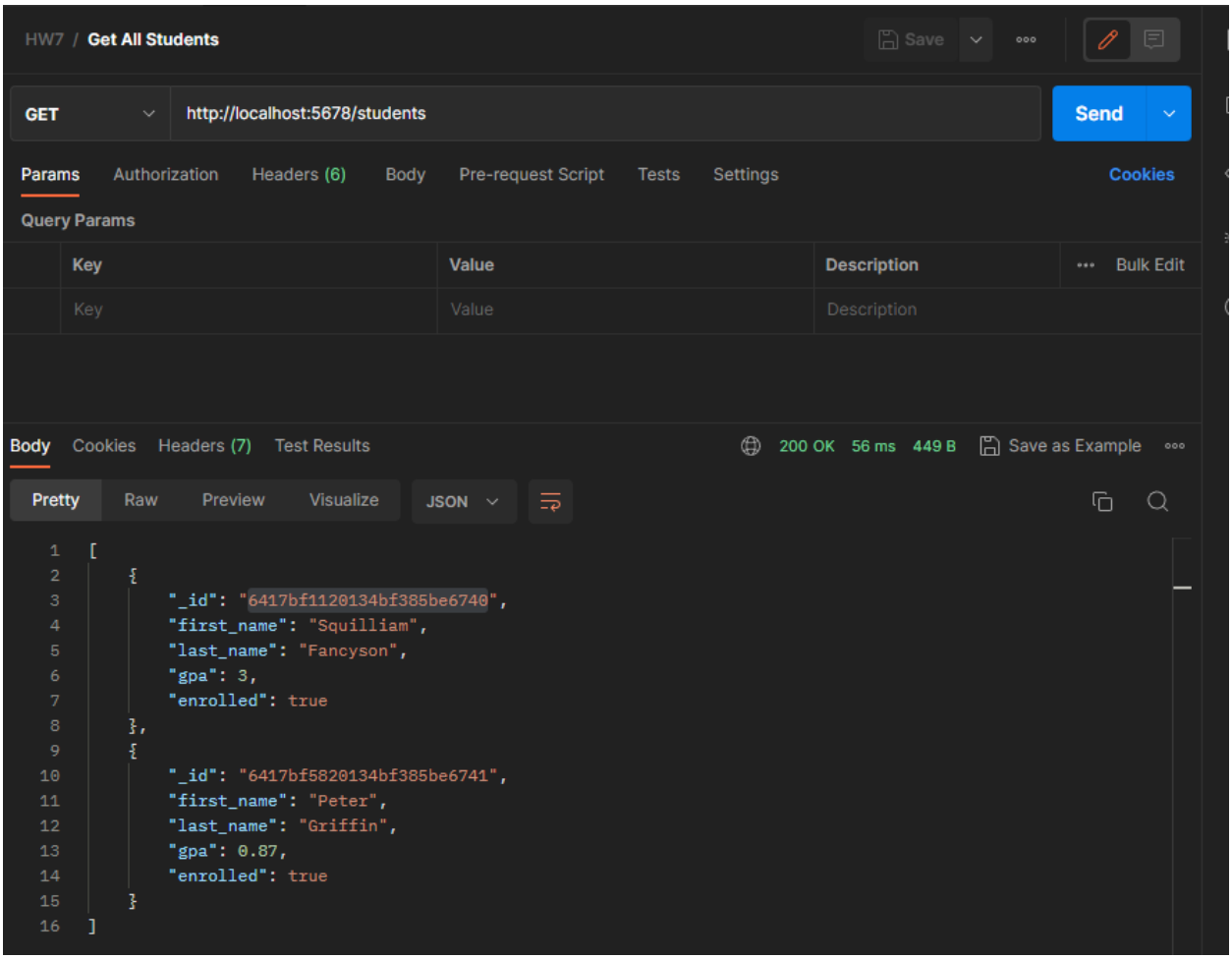
4. GET: Get a Single Student:

   I need Squilliam's GPA, but I can't remember what it is, so I used the GET request to find out the information about him. I used his record ID as the endpoint, and found Squilliam's data as a result.



5. GET: Get All Students:

   Finally, I wanted to make sure both of my students, Squilliam and Peter, were still present in the database. To do this, I used the GET request with /students as the endpoint, to simply see all students in the database.



FRONT END CHECK

Now that I knew my server and MongoDB were interacting with each other properly, I needed to make sure my front end worked for submitting requests, not just Postman. Luckily, everything seemed to work.

Home Add Update Delete Lookup List All

Homework 7 Home Page

William Reed

Z23564142

Menu

POST: Add Student

PUT: Update Student

DELETE: Delete Student

GET: Display Student

GET: List Students

Add a Student

First Name: Billy

Last Name: Bob

GPA: 5.6

Enrolled? Yes

Add Student

Student created with ID: "6417c2c920134bf385be6743"

Update a Student's Data

Record ID: 6417c2c920134bf385be67

First Name: Billy

Last Name: Bob

GPA: 2.5

Enrolled? Yes

Update Student

Student with ID: 6417c2c920134bf385be6743 Updated Successfully

Delete a Student

Record ID: 6417c2c920134bf385be67

Delete Student

Student with ID: 6417c2c920134bf385be6743 Deleted.

Student Lookup

Enter student ID: 6417bf1120134bf385be67

Lookup

First Name	Last Name	GPA	Enrolled	ID
Squilliam	Fancyson	3	Enrolled	6417bf1120134bf385be6740

Students

First Name	Last Name	GPA	Enrolled?	Student ID
Squilliam	Fancyson	3	true	6417bf1120134bf385be6740
Peter	Griffin	0.87	true	6417bf5820134bf385be6741