Module 3 Lab 2: Custom Modules

Kayana L. Wenglarz

Arizona State University

IFT458: Middleware Prog & Database Sec

Prof. Dinesh Sthapit

September 24, 2023

Module 3 Lab 2: Custom Modules

Code Analyzation

After reviewing both the before and after of the provided code sets, the SRP principle was clearly being violated. The SRP principle states that "A class should have one, and only one, reason to change." This principle was introduced by Robert Martin and is a key component to the SOLID principles in software. Within the app.js files, we have the application setup, the routes, and even some of the application logic built within this file. Putting too much application logic inside of one file breaks SRP and writing programs this way makes them hard to maintain, work on, and review/make changes. With such a small server application, the files are already becoming dense, and it's hard to keep track of where certain logic is placed. It is best practice to split the work into separate modules so that each module in the application has a distinct and clear purpose. This makes code readable, easier to maintain, and follows best practices for building large programs. To do this, I will follow best practices and refactor the code with the SRP principle in mind.

Figure 1

*To start, I created new directories and files to clean up the project. There are two high-level directories each appended with P**artA** and P**artB**. Part A holds the code from the first downloaded code set - I left this unchanged. **Part B** contains the code set from the second download. I have updated the project through some refactoring.*

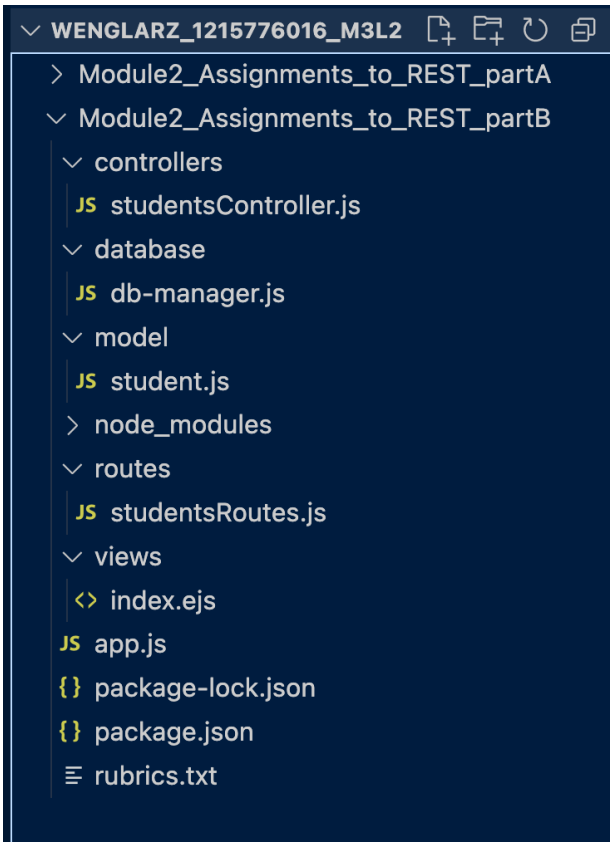


Figure 2

I started off simple: I created the basic model for a student so that we have a class that will allow us to create Student objects.

```
JS student.js ×
Module2_Assignments_to_REST_partB > model > JS
1   class Student {
2       id;
3       firstName;
4       middleName;
5       lastName;
6       address;
7       email;
8       phone;
9       description;
10  }
11
12  module.exports = Student;
```

Figure 3

I extracted most of the non-application-setup logic from app.js to other files that I will cover below. Now, app.js contains only the application setup and listens on the port. This is now the entry point of our application and contains much less cruft. The idea behind this is if there is any reason to update the setup of the application (body parsing, the port number to spin up the server on, etc.), we will only need to update this file. Please note that I had to add line 10 so that our application can grab the JSON from the Postman requests we will send later during the testing. I also added lines 20-22 to grab the newly created routes file.

```
JS app.js ×
Module2_Assignments_to_REST_partB > JS app.js > ...
// Import the required modules
const express = require('express');
const bodyParser = require('body-parser');
const path = require('path');

// Create an instance of express
const app = express();

// HAD TO ADD - need to allow express to parse out JSON body
app.use(express.json());

// We use the 'body-parser' middleware to parse the incoming request bodies
app.use(bodyParser.urlencoded({ extended: false }));

// Set the view engine to ejs
app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));
console.log('views', path.join(__dirname, 'views'));

// Add the routes to the app
const studentsRouter = require('./routes/studentsRoutes');
app.use('/students', studentsRouter);

// Start the server on port 3000
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Figure 4

Next, I created a new studentRoutes.js file that holds all of the routes for our API. If we ever need to update any of the routes, or add more, we only need to update one file. In the file, the routes point to the functionality setup in the studentsControllers.js file that I have created.

```
JS studentsRoutes.js ×
Module2_Assignments_to_REST_partB > routes > JS studentsRoutes.js > ...
1   const express = require('express');
2   const router = express.Router();
3   const studentsController = require('../
    controllers/studentsController');
4
5   router
6       .route('/')
7       .get(studentsController.getAllStudents)
8       .post(studentsController.addStudent)
9
10  router
11      .route('/:id')
12      .get(studentsController.getStudentByID)
13      .put(studentsController.updateStudentByID)
14      .delete(studentsController.deleteStudentByID)
15
16  module.exports = router;
17
```

Figures 5 & 6

This brings us to the studentsControllers.js file I created to hold the refactored middleware logic in our application. Each of the functions from the Router setup above is created here. The actual "communication" level with the students database here is refactored into the db-manager.js file that I created in the database directory. Within the controllers file, I had to update some of the code to make the endpoints work as expected for our application - the changes can be seen in the file below, but we can now see that if we need to update any of the middleware logic in our application, this is the place to do so.

studentsController.js

Module2_Assignments_to_REST_partB > controllers > studentsController.js > deleteStudentByID

```javascript
const dbManager = require('../database/db-manager');

exports.getAllStudents = function(req, res) {
    // Render the form and pass in the current student data
    res.render('index', { students: dbManager.getStudents() });
};

exports.addStudent = function(req, res) {
    // Add the submitted student data to our data store
    dbManager.addStudent(req.body);

    // Redirect back to the form
    res.redirect('/students');
}

// The GET a specific route for the form students data base
exports.getStudentByID = function(req, res) {
    // Render the form and pass in the current student data
    const id = numeric(req.params.id);
    res.render('index', { students: dbManager.getSpecificStudent(id) });
};

exports.updateStudentByID = function(req, res) {
    // Parse ID
    const id = Number(req.params.id);

    // Add the submitted student data to our data store
    dbManager.upDateStudent(id, req.body);

    // Redirect back to the form
    res.redirect('/students');
};

exports.deleteStudentByID = function (req, res) {
    // Parse ID
  const id = Number(req.params.id);

  // Delete the student
  dbManager.deleteStudent(id);
```

```
JS studentsController.js ×
Module2_Assignments_to_REST_partB > controllers > JS studentsController.js > delete
17  exports.getStudentByID = function(req, res) {
18      // Render the form and pass in the current student data
19      const id = numeric(req.params.id);
20      res.render('index', { students: dbManager.getSpecificStudent(id) }
21  };
22
23  exports.updateStudentByID = function(req, res) {
24      // Parse ID
25      const id = Number(req.params.id);
26
27      // Add the submitted student data to our data store
28      dbManager.upDateStudent(id, req.body);
29
30      // Redirect back to the form
31      res.redirect('/students');
32  };
33
34  exports.deleteStudentByID = function (req, res) {
35      // Parse ID
36    const id = Number(req.params.id);
37
38    // Delete the student
39    dbManager.deleteStudent(id);
40
41    // Redirect back to the form
42    res.redirect('/students');
43  }
```

Figures 7 & 8

The db-manager.js is file I created is responsible for interacting with our datastore (in this case, the students array). We import the Student model that we have created, and I updated most of the functions to work with our application. The manager now has 5 distinct functions that are used within each route/controller setup

1. *addStudent – This is called when the application wants to add a user (POST).*
2. *getStudents - This is called when the application wants to grab all the students (GET).*

3. *getSpecificStudent - This is called when the application wants to grab a specific student (GET).*
4. *upDAteStudent - This is called when the application updates a student (PUT).*
5. *delSpecificStudent - This is called when the application deletes a student by id (DELETE).*

db-manager.js

Module2_Assignments_to_REST_partB > database > db-manager.js > ...

```javascript
//This is where we will implement our CRUD operations.
const Student = require('../model/student');

// Create a data store for our student data
let students = [];

// create a new student
addStudent = function (student) {
    const newStudent = new Student();
    newStudent.id = students.length + 1;
    newStudent.firstName = student.firstName;
    newStudent.middleName = student.middleName;
    newStudent.lastName = student.lastName;
    newStudent.address = student.address;
    newStudent.phone = student.phone;
    newStudent.email = student.email;
    newStudent.description = student.description;

    students.push(newStudent);

    return newStudent;
};

//update specific student
upDateStudent = function (id, student) {
    const specificStudent = students.find(student => student.id === id);

    if(specificStudent) {
        const updatedStudent = new Student();
        updatedStudent.id = student.id;
        updatedStudent.firstName = student.firstName;
        updatedStudent.middleName = student.middleName;
        updatedStudent.lastName = student.lastName;
        updatedStudent.address = student.address;
        updatedStudent.phone = student.phone;
        updatedStudent.email = student.email;
        updatedStudent.description = student.description;

        students[students.indexOf(specificStudent)] = updatedStudent;
```

```
JS db-manager.js M ×
Module2_Assignments_to_REST_partB > database > JS db-manager.js > ...
34            updatedStudent.address = student.address;
35            updatedStudent.phone = student.phone;
36            updatedStudent.email = student.email;
37            updatedStudent.description = student.description;
38
39            students[students.indexOf(specificStudent)] = updatedStudent;
40        }
41    };
42
43    // get all students
44    const getStudents = function () {
45        return students;
46    }
47
48    // get a specific student
49    getSpecificStudent = function (id) {
50        const specificStudent = students.find(student => student.id === id);
51        if(specificStudent){
52            return specificStudent;
53        }
54        return undefined;
55    }
56    // delete a specific student
57    delSpecificStudent = function (id) {
58        const specificStudent = students.find(student => student.id === id);
59        if(specificStudent){
60            const index = students.indexOf(specificStudent);
61            students.splice(index, 1) ;
62        }
63    }
64
65    exports.getSpecificStudent = getSpecificStudent;
66    exports.deleteStudent = delSpecificStudent;
67    exports.getStudents = getStudents;
68    exports.upDateStudent = upDateStudent;
69    exports.addStudent = addStudent;
70    exports.students = students;
```

Figure 9

The index files holds the logic to render the data to the user. I made some changes, for example, I updated the student variable attributes, so that we can appropriately display our data.

```
index.ejs
Module2_Assignments_to_REST_partB > views > index.ejs > html > body > ? > ? > a
<!DOCTYPE html>
<html>
<head>
  <title>Student Information Form</title>
</head>
<body>
  <!-- This is the form for the user to input their data -->
  <!-- When the form is submitted, it sends a POST request to the server at the specified action URL ("/") -->
  <form action="/students" method="POST">
    <label for="firstName">First Name:</label><br>
    <input type="text" id="firstName" name="firstName"><br>
    <label for="middleName">Middle Name:</label><br>
    <input type="text" id="middleName" name="middleName"><br>
    <label for="lastName">Last Name:</label><br>
    <input type="text" id="lastName" name="lastName"><br>
    <label for="address">Address:</label><br>
    <input type="text" id="address" name="address"><br>
    <label for="phone">Phone:</label><br>
    <input type="tel" id="phone" name="phone"><br>
    <label for="email">Email:</label><br>
    <input type="email" id="email" name="email"><br>
    <label for="description">Description:</label><br>
    <textarea id="description" name="description"></textarea><br>
    <input type="submit" value="Submit">
  </form>

  <br>

  <!-- This is where we'll display the submitted student data -->
  <!-- EJS allows us to use JavaScript in our HTML. Here, we loop over the 'students' array and create a new list item for each student -->
  <% if(students !== undefined) { %>
    <ul>
      <% students.forEach(function(student) { %>
        <li>
          <h2>Name: <%= student.firstName %> <%= student.middleName %> <%= student.lastName %></h2>
          <p>Address: <%= student.address %></p>
          <p>Phone: <%= student.phone %></p>
          <p>Email: <%= student.email %></p>
          <p>Description: <%= student.description %></p>
        </li>
      <% }); %>
    </ul>
  <% }; %>

  <!-- This is a link that allows the user to add more student data -->
  <!-- When clicked, it sends a GET request to the server at the specified URL ("/") -->
  <a href="/">Add New Student</a>
```

Figure 10

Now that the code has been refactored, I was ready to start testing the application. The first test I ran against my server was a simple GET test with the http://localhost:3000/students route. In Postman, I see the response from the server.

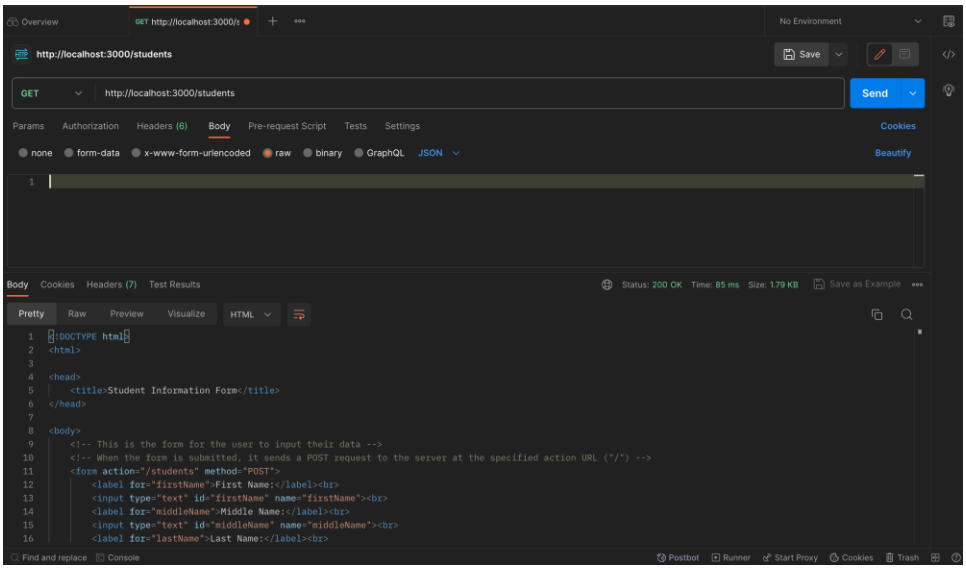


Figure 11

This is the response rendered in the browser.

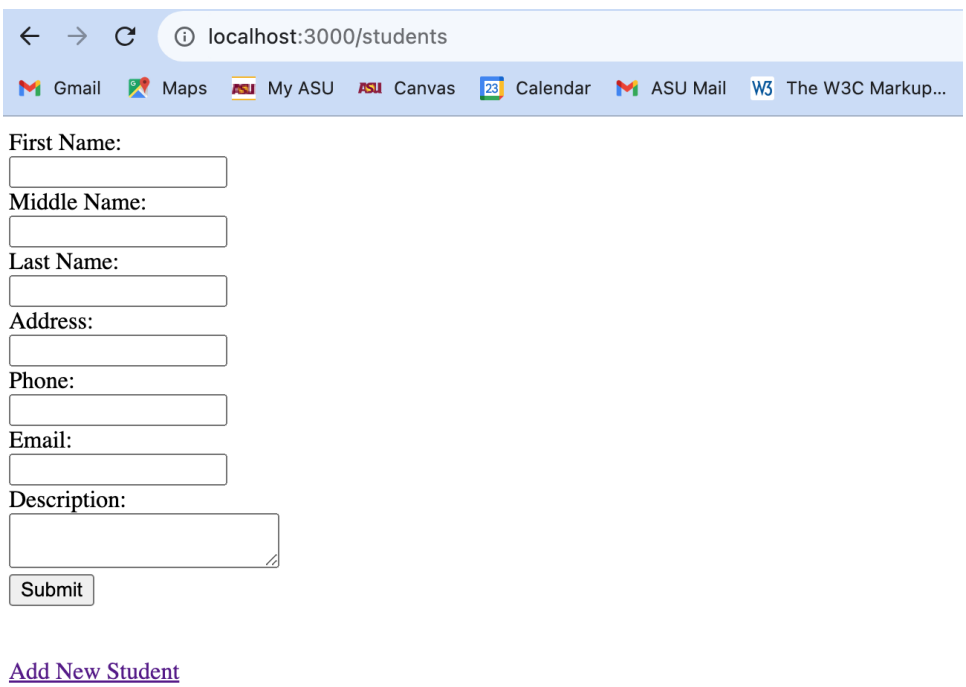


Figure 12

Next, I tested to see if I could POST a new student. In Postman, I hit the http://localhost:3000/students route with a POST method. The request body of my response was the following JSON object:

{

"firstName": "Kayana",

"middleName": "L",

"lastName": "Wenglarz",

"address": "520 E Weddell Dr",

"phone": "217-402-6712",

"email": "kwenglar@asu.edu",

"description": "My user"

}

When hitting the route in Postman, we see the following response with the newly added student in the data.

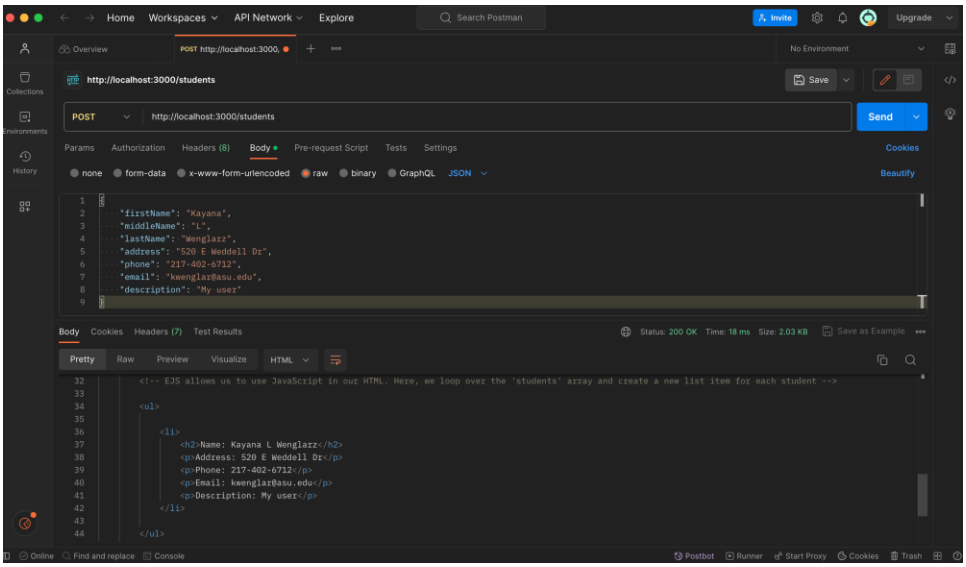


Figure 13

With a simple refreshing of the page in the browser, we can see the updated page.

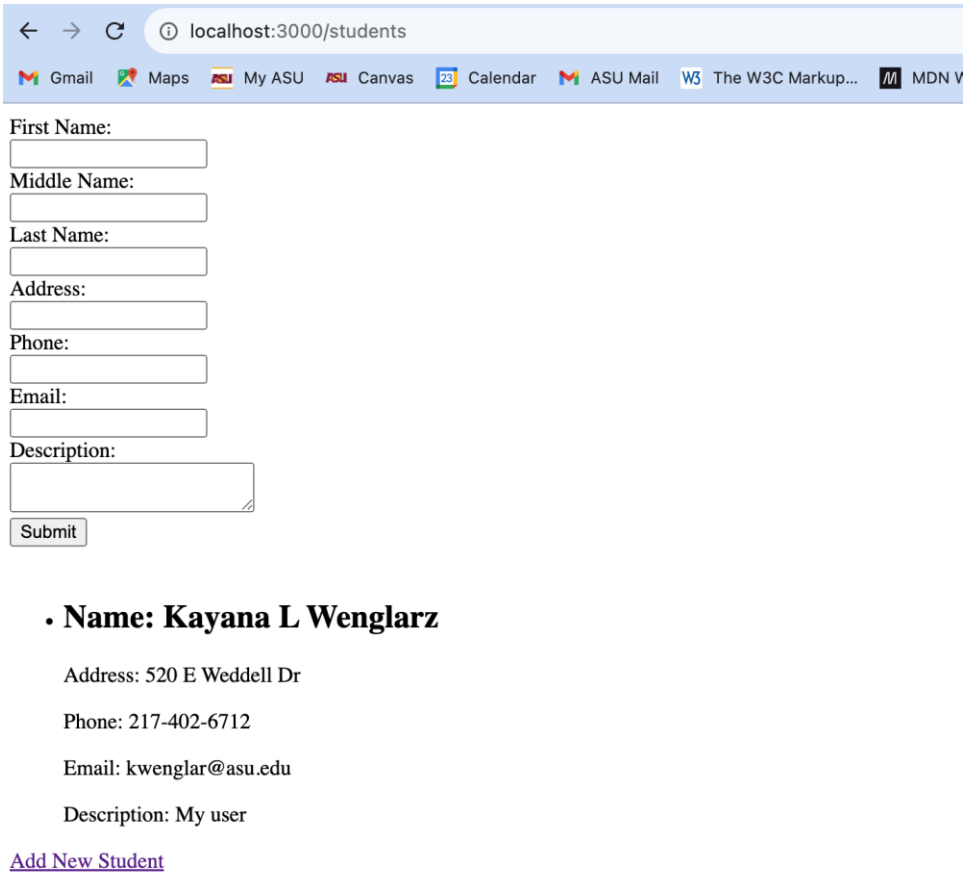


Figure 14

The next step is to test the PUT route to update the student we created above. The previously created student has id = 1, so the route I will hit will be http://localhost:3000/students/1. I had to update the code below to include the ID in the route, parse out the ID from the route, and send the ID (along with the request body, the student info) to the updateStudent method. Then I simply redirect back to the "/students" route.

```
exports.updateStudentByID = function(req, res) {
    // Parse ID
    const id = Number(req.params.id);

    // Add the submitted student data to our data store
    dbManager.upDateStudent(id, req.body);

    // Redirect back to the form
    res.redirect('/students');
};
```

Figure 15

I then updated the updateStudent method to appropriately grab the old student, and if it exists, it will update the student accordingly.

```
//update specific student
upDateStudent = function (id, student) {
    const specificStudent = students.find(student => student.id === id);

    if(specificStudent) {
        const updatedStudent = new Student();
        updatedStudent.id = student.id;
        updatedStudent.firstName = student.firstName;
        updatedStudent.middleName = student.middleName;
        updatedStudent.lastName = student.lastName;
        updatedStudent.address = student.address;
        updatedStudent.phone = student.phone;
        updatedStudent.email = student.email;
        updatedStudent.description = student.description;

        students[students.indexOf(specificStudent)] = updatedStudent;
    }
};
```

Figure 16

I attempted to update the created student's name to Margot Elise Robbie, I didn't change anything else for simplicity's sake. We then send the following data through the request body in Postman and we see the response in Postman with the updated user. The address, phone, and email all stayed the same, while the name attributes and description were updated accordingly.

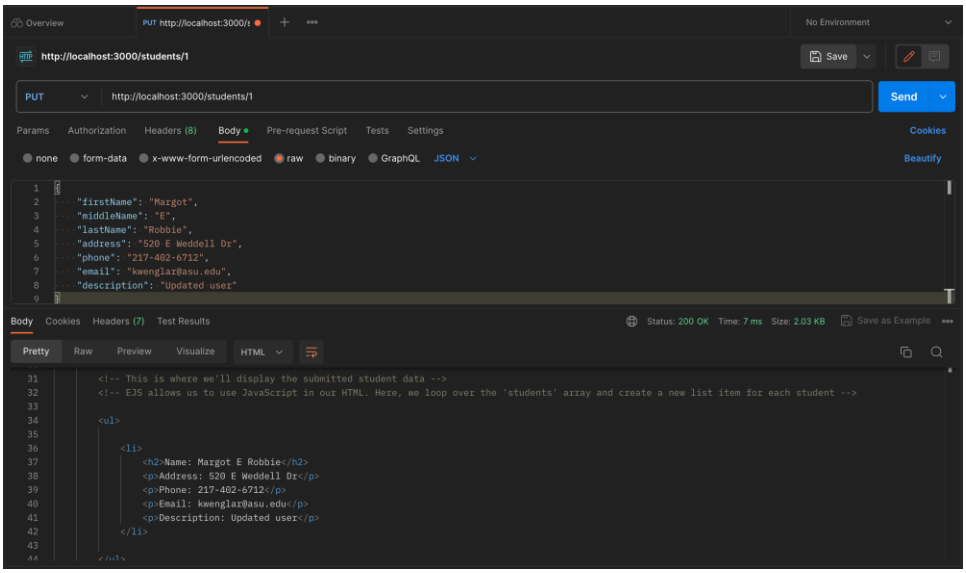


Figure 17

I refreshed the browser to see the results there as well.

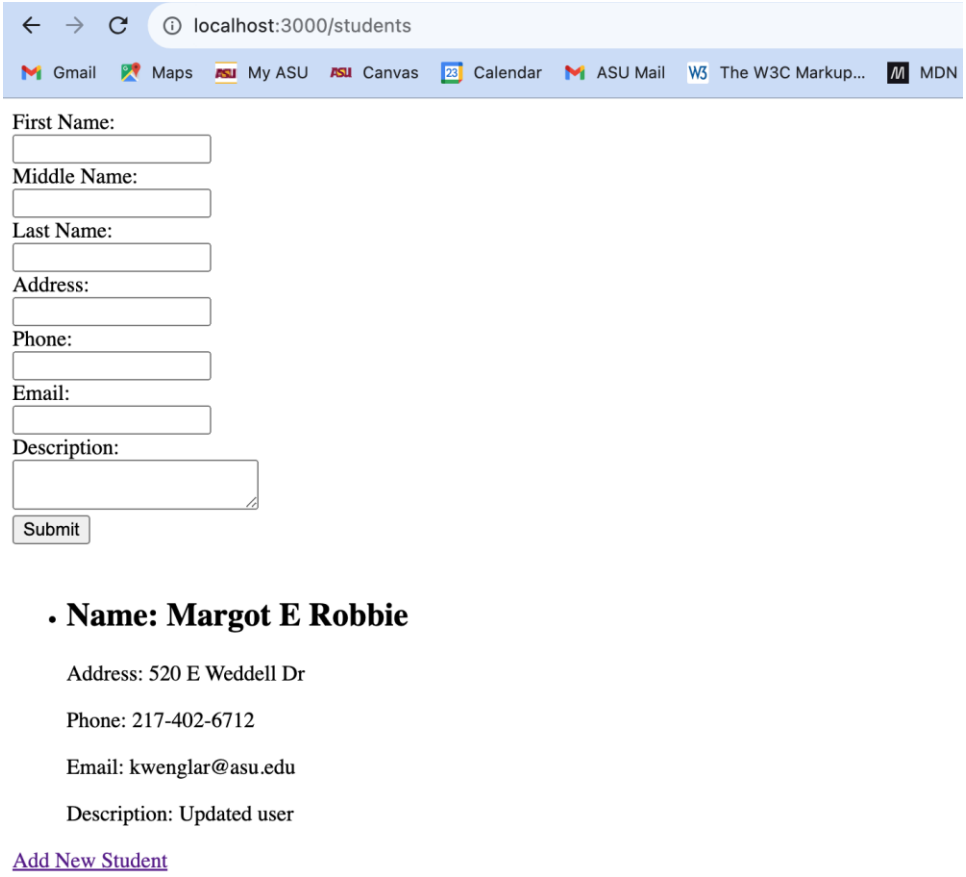


Figure 18

The last route to test was the Delete route. I had to update the route behavior similar to the PUT route. The code was updated to parse the id, call the deleteStudents method on the students module with the id, and then render the "/students" route.

```
exports.deleteStudentByID = function (req, res) {
    // Parse ID
  const id = Number(req.params.id);

  // Delete the student
  dbManager.deleteStudent(id);

  // Redirect back to the form
  res.redirect('/students');
}
```

Figure 19

I also updated StudentsController deleteStudents function to access the student with that id and update the students accordingly.

```
// delete a specific student
delSpecificStudent = function (id) {
    const specificStudent = students.find(student => student.id === id);
    if(specificStudent){
        const index = students.indexOf(specificStudent);
        students.splice(index, 1) ;
    }
}
```

Figure 20

We can now use Postman to hit the route with DELETE method. We see the response with the updated students array as empty (we deleted the added user), so we can see it was successful.

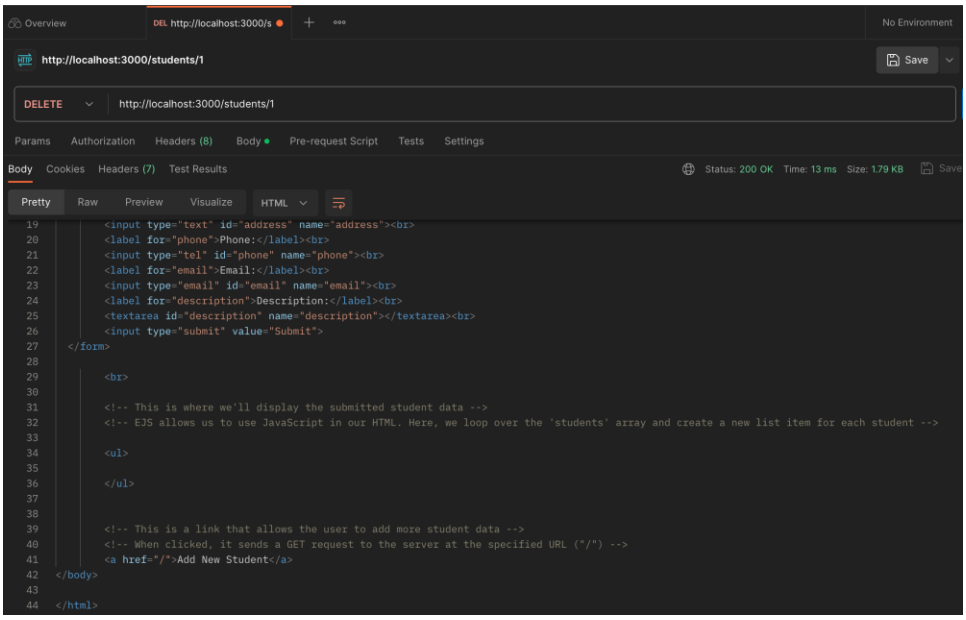
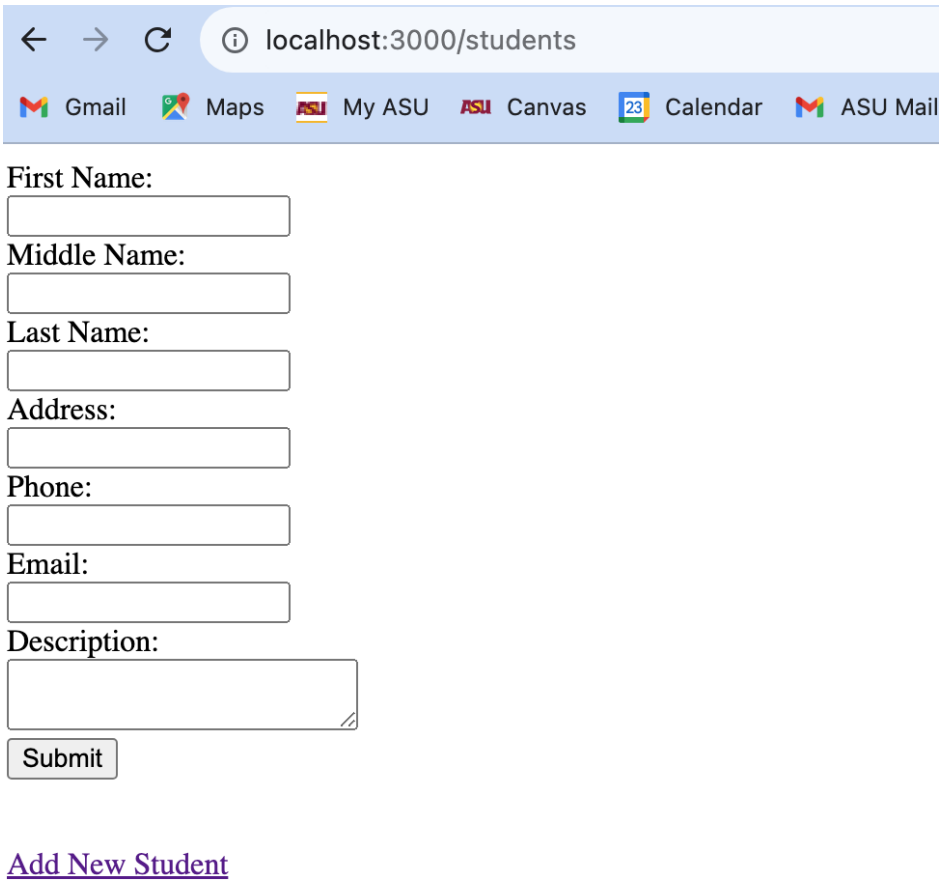


Figure 21

Next, I refreshed the browser to see there are now no students in the array.



Summary

Before making the changes, the code was becoming overcrowded. The files had more than one responsibility (clear violation of the SRP principle), and it was hard to follow in general. With the newly refactored code, we can see the project is now adhering to the SRP principle. If we need to update anything regarding the routes, we have one single module studentsRoutes to update. If we need to update how we are interacting with our datastore, we update the db-manager module. Every file serves a specific purpose now, and this allows the project to be more maintainable in the long term. In the real world, code bases can get very large.

It is important to use best practices so the project has a clear structure and is comprehensive to programmers.