CIS 550: Database and Information Systems

Homework 2 MS 1: Building an API

(40 points)

Please read this handout from start to finish before proceeding to work on the assignment

Introduction

In this two-part assignment, you will incrementally build an interactive FIFA-themed web application using React and Node.js backed by a MySQL RDS database.

To help you understand the fundamentals of web interface design and implementation, in this first milestone (MS1, worth 40 points) you will build an API on Node.js following the specification that we lay out for you (outlined below). This will help you understand how APIs are able to process and facilitate data interchange between various (client) applications and databases.

In the next milestone (MS2, worth 60 points), you will build a React frontend that uses this API.

Advice to Students

We expect you to already be familiar with web development (since it is a prerequisite to CIS 550). If you are relatively inexperienced, you will find that these two milestones have a steep learning curve; please attend recitation and complete Exercise 3 to prepare you for this homework. You should also pay careful attention to all the details in the specifications.

This assignment is crucial for you to understand many external applications of databases and will provide you with the opportunity to develop many important qualities and skills that you need as a software developer like learning to read documentation, understanding and implementing specifications, and debugging. You will also need to know some very basic (npm and) terminal operations and basic endpoint routing in Node.js.

While the SQL queries for this assignment are intentionally made to be very easy, we would like you to get hands-on experience with the web-development related aspects.

If you are already familiar with version control (Git), we recommend you use it to regularly commit and save your work. Since this assignment also serves as a precursor to the term project in many ways, even if you are not familiar with Git, you should try to learn some basics. There are plenty of resources available online that can help you learn. This is only a recommendation and is in no way necessary but we encourage you to learn now. Either way, please be sure to save your work from time to time.

Please start early, be patient, and avoid last minute Piazza and OH traffic!

Setup

Required

You will need the latest version of Node.js on your machine for this assignment. You should verify that the following commands run and give a reasonable output on your terminal:

`npm -v`
`node - v`

The recommended Node version is 14.17.x, where x can be any number, but older versions of Node would probably work as well. If you have problems with older Node versions, you should update!

You will also need to use the (built-in) terminal for your operating system and should have a code editor (with the ability to open *.md*, *.js*, and *.json* files)

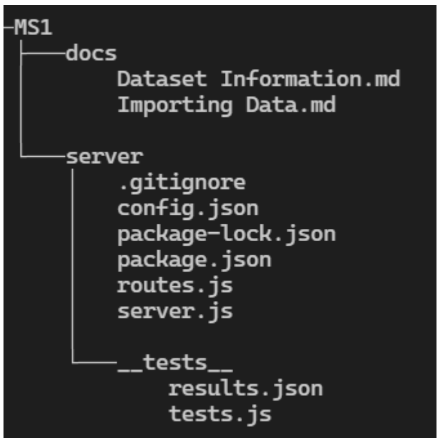
Recommended

We highly recommend that you use Visual Studio (VS) Code (as the code editor and its built in terminal). You will also find its built in .md preview feature very helpful in reading some documentation files (which are in *.md* format)

You will also need a web browser with a developer console (highly recommended for testing your API). We recommend Google Chrome, but most major browsers will support equivalent functionality.

Application Structure

Unzip HW2-MS1.zip from the assignment page. You will see that the file directories are as follows:



```
-MS1
 ├──docs
 │      Dataset Information.md
 │      Importing Data.md
 │
 └──server
        .gitignore
        config.json
        package-lock.json
        package.json
        routes.js
        server.js
     │
     └──__tests__
            results.json
            tests.js
```

Here is an explanation of these folders and their respective files:

/docs

This folder contains the required documentation you need to follow for the homework. Specifically:

- Dataset Information.md: Refer to this file and the listed sources in conjunction with *Importing Data.md* for information on the encodings, abbreviations, and types used in the datasets.

- Importing Data.md: This document provides information on importing the datasets into your RDS instance. It also contains the DDL statements that you will need to run to create the tables. The schema of the tables can be inferred from these DDL statements.

/server
This folder holds the server application files, tests, and dependencies (as required by Node.js).

- .gitignore: A gitignore file for the Node application. Read more on .gitignore files here.
- config.json: Holds the RDS connection credentials/information and application configuration settings (like port and host).
- package.json: maintains the project dependency tree; defines project properties, scripts, etc
- package-lock.json: saves the exact version of each package in the application dependency tree for installs and maintenance.
- routes.js: This is where the code for the API routes' handler functions go. We have already defined the necessary routes for you - follow the *'TODO:'* comments and implement/modify them as specified). This is the only file that you should need to modify since it is the only one you will submit. Your routes.js must be compatible with the other files that we provide, so do not update anything else
- server.js: The code for the routed HTTP application. You will see that it imports *routes.js* and maps each route function to an API route and type (like GET, POST, etc). For this HW, we will only use GET requests. It also 'listens' to a specific port on a host using the parameters in *config.json*.

/__tests__
This folder contains the test files for the API:

- results.json: Stores (some) expected results for the tests in a json encoding.
- tests.js: Stores

Getting Started

Make sure that you have the required software installed and have a good understanding of the lecture and recitation materials before proceeding.

Open the unzipped HW2-MS1 folder. If you are using VS Code, you should be able to do this by clicking *File -> Open Folder* from the top menu. You could also just use the 'code' command on the terminal or right click on the folder and select 'open with code' if you have added VS code to the terminal or to the options menu for your system respectively.

Open a new terminal (on VS code) and cd into the server folder, then run npm install:

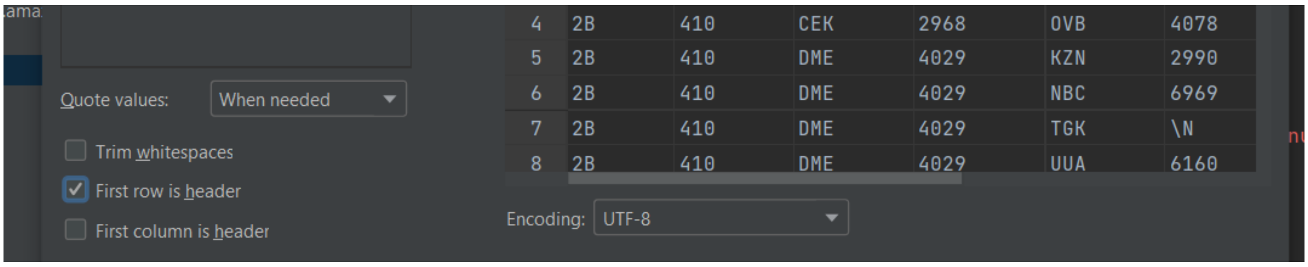
```
cd server
npm install
```

This will download and save the required dependencies into the *node_modules* folder within the server directory.

Importing Data

Set up a MySQL instance on AWS RDS (allow in and outbound traffic of ‘All Types’ from ‘Anywhere’ for the instance, not just from ‘My IP’). Delete any other security group rules. Connect to the database using DataGrip (as outlined in the DataGrip handout from HW1). Open a new query execution console and use the DDL statements in *Importing Data.md* to create two tables, Players and Matches, in a database named ‘FIFA’.

As mentioned in *Importing Data.md,* please ensure that while importing the data, you have the ‘First row is header’ option on the import wizard checked as shown below!



Quote values: When needed

Trim whitespaces

[✓ First row is header]

First column is header

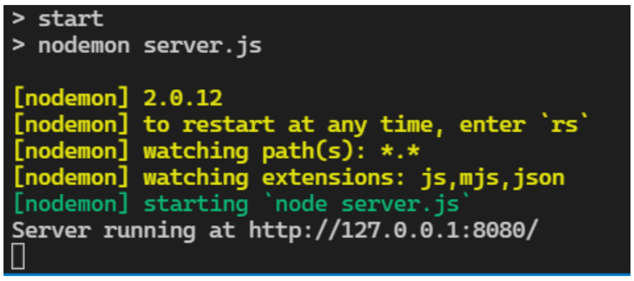
4	2B	410	CEK	2968	OVB	4078
5	2B	410	DME	4029	KZN	2990
6	2B	410	DME	4029	NBC	6969
7	2B	410	DME	4029	TGK	\N
8	2B	410	DME	4029	UUA	6160

Encoding: UTF-8

<u>Fill in the db credentials into config.json in the server folder</u>

Understanding API Routes

Recitation 3 and the lecture materials provide some great guidance on this, but here is a summary of how routes work on a server. First, you should start the server application by running the command **`npm start`** in a terminal window and follow along as needed. The following output confirms that the server is running on localhost:



```
> start
> nodemon server.js

[nodemon] 2.0.12
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Server running at http://127.0.0.1:8080/
```

The server application accepts incoming HTTP requests by ‘listening’ on a specified port on a host machine. For example, this application (server.js) runs on the host ‘localhost’ and port 8080 as specified using the configuration file (*config.json*).

Upon receiving a request from a client, the server application parses the URL string to map it to a registered route handler, extracting important information including the route and query parameters. For example, a request to ‘http://localhost:8080/hello’ from your browser will use the GET route registered on the server application, map it to the route handler function **`hello(req, res)`** in *routes.js*, (this is Route 1 in the code) and return the string `"Hello! Welcome to the FIFA server!"` using the res.send function. Here, **`req`** maps to the request object and **`res`** the response object.

You should look closely at these parts of server.js and routes.js. to confirm and consolidate your understanding, and note that the behaviour of the route is different when certain [query parameters]() are added to this URL. Specifically, the query parameter **`name`** for this route can be accessed via the request object as `req.query.name`. The route handler first checks if there is such a query parameter (`name`), and if so, returns the string `"Hello! ${req.query.name}, welcome to the FIFA server!"`. For example, a request to http://localhost:8080/hello?name=Steve would return the message: `"Hello, Steve! Welcome to the FIFA server!"`

With this understanding, let's move on to a short warm up exercise before we implement the routes directly from the specification.

A Warm Up

(4 Points)

In this section, you will fill in the code for a route that is already defined in routes.js, but isn't quite implemented to the exact specification. This will serve as a great starting point for understanding the spec terminology, instructions, tests, and the related debugging processes.

TASK 1

Look at the function `jersey` in routes.js, which is the route handler for the GET route `/jersey/:{choice}`. The specification for this route, in plain english, is as follows:

Route 2: `/jersey/:{choice}`
Route Parameter(s): `choice` (string)
Query Parameter(s): `name` (string)* (default: `"player"`)
Route Handler: `jersey(req, res)`
Return Type: JSON
Return Parameters: { `message` (string), `jersey_number` (int)*, `jersey_color` (int)* }
Expected (Output) Behaviour:

- Case 1: If the route parameter (`choice`)= 'number'
  - Return { message: _ , jersey_number: _ }
  - Where message = "Hello, `name`!"
  - And jersey_number is (an integer) in the range from 1 to 20, both included
- Case 2: If the route parameter(`choice`)= 'color'
  - Return { message: _ , jersey_color: _ }
  - Where message = "Hello, `name`!"
  - and jersey_color = either 'red' or 'blue' (returned at random, don't return just one color all the time)
- Case 3: If the route parameter is defined but does not match cases 1 or 2:
  - Return { message: _ }
  - Where message = "Hello, `name`, we like your jersey!"

NOTE: Parameters, unless specified, are required. Optional parameters are marked with an asterisk (*). Default values are indicated when necessary. You will find these links helpful in understanding route and query parameters.

Now, take a look at a **buggy** implementation of the route handler we provide, **`jersey(req, res)`** in *routes.js*. You might immediately catch some bugs since you are able to look at the code that implements this function, but here is how you could 'see it in action' while debugging:

Method 1 (Inspecting responses through a browser)
After starting the server application, open Google Chrome (or any web browser). Let's first test the route for Case 1, which refers to the case where the route parameter (**`choice`**) is 'number'.

There are two 'sub' cases to test here in terms of behaviour (the optional query parameter (**`name`**) and its default value). Head over to the following two links on your web browser and inspect the output against the spec:

- http://localhost:8080/jersey/number
- http://localhost:8080/jersey/number?name=Steve

You will quickly observe that except for one of the return parameters being named incorrectly, there seems to be nothing wrong in the results or the implementation of this route. Specifically, the route returns **{ message: _, lucky_number: _}** instead of **{ message: _, jersey_number: _}**.

This should be an easy fix, but let's think about another detail once you correct the implementation by modifying the code under the comment `// TODO: TASK 1: inspect for issues and correct`. How do we ascertain (from just assessing responses from the browser) that the route always returns a random number in the range [1, 20]? In fact, what if:

- The number is always just the same (say, 4)?
- The number is somehow dependent on an input instead of being really independent (say, it returns 5 if the length of the query parameter **`name`** is > 2)

One could very well argue that it is (somehow) possible to make multiple browser requests for a well chosen set of tests and identify errors (and error patterns) like this, but let's look at another way to debug!

TASK 2

Method 2 (Unit testing)
Ensure that the server is not currently running (to close the server process if it's running on a terminal, use Command + C on a Mac or Ctrl + C on Windows). You can run the provided tests in the __test__ directory within the server folder by running:

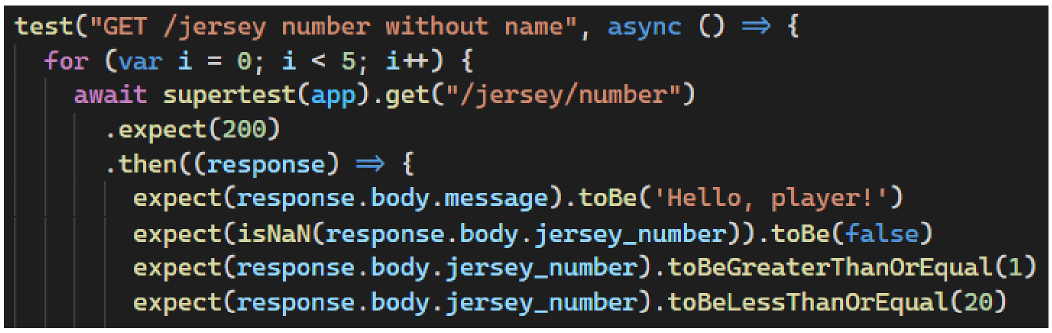
```
npm test
```

You will see 2 passing tests for this route (labelled `GET /jersey number without name` and `GET /jersey number with name`), but there will also be 3 failing tests for this route at this point:

- `GET /jersey color without name`
- `GET /jersey color with name`
- `GET /jersey other value without name`

You will also see other failing tests, but those are not for this route, so don't worry about them now!

Observe that in the list above, the first two cases correspond to Case 2, which is when the route parameter (**choice**) is 'color'. Let's look at the first test case (`GET /jersey color without name`), which is contained within the following function in tests.js:



```
test("GET /jersey number without name", async () => {
  for (var i = 0; i < 5; i++) {
    await supertest(app).get("/jersey/number")
      .expect(200)
      .then((response) => {
        expect(response.body.message).toBe('Hello, player!')
        expect(isNaN(response.body.jersey_number)).toBe(false)
        expect(response.body.jersey_number).toBeGreaterThanOrEqual(1)
        expect(response.body.jersey_number).toBeLessThanOrEqual(20)
```

A quick examination of the test case reveals that it samples from 5 test responses (think about why it is necessary for this route), checking various attributes. The fix should be simple enough (follow the TODO comment: `// TODO: TASK 2: change this or any variables above to return only 'red' or 'blue' at random (go Quakers!)`), but if one would simply open a browser window, two-thirds of the time, they'd see no issues with the response!

Here are two additional observations:

1. Once you fix this, you will also pass the test `GET /jersey color with name`. You might fail multiple test cases because of a single error!
2. The test cases we provide check for many different things at once since we are testing for overall functionality. It is technically possible to make them more specific (and we encourage you to write your own tests if you'd like, although we won't grade them).

Our test cases are in no way exhaustive, and the test cases we provide are worth 20 points. See the section on Submission and Grading for more details.

TASK 3

Let's finish up this warm up exercise with a final illustration. Recollect that we still fail the test `GET /jersey other value without name`. Checking the response for (http://localhost:8080/jersey/xyz) on the browser response or the test might reveal no issues to many, but actually, you will see that the response is missing a space between 'Hello,' and 'player' and hence does not match the specification!

If you missed this before, we're sure that a second look at either the browser response or the test case will make a lot more sense. If you haven't already, correct this error.

Try to follow the spec as closely as possible, and use a mixture of debugging and testing techniques to ensure that you are indeed on the right track!

API Specification

(36 points)

In this section, you will complete the following 6 routes whose english specification is given below. Each route is worth 5-7 points, and their skeleton (which you must complete) can be found in routes.js.

Note that the information to be output is specified in the return parameters. You should consult ***Data Information.md*** to understand for more insight into the data like abbreviations used in the column names and the data.

General Routes

TASK 4

Route 3: `/matches/:{league}`

Description: Returns an array of selected match attributes for a particular league sorted by home team first then the away team - both in alphabetical order
Route Parameter(s): **`league`** (string)
Query Parameter(s): **`page`** (int)*, **`pagesize`** (int)* (default: `10`)
Route Handler: **`all_matches(req, res)`**
Return Type: JSON
Return Parameters: {**`results`** (JSON array of { **`MatchId`** (int), **`Date`** (string), **`Time`** (string), **`Home`** (string), **`Away`** (string), **`HomeGoals`** (int), **`AwayGoals`** (int)}) }
Expected (Output) Behaviour:

- Case 1: If the page parameter (**page**) is defined
    - Return match entries with all the above return parameters for that page number by considering the **`page`** and **`pagesize`** parameters. For example, page 1 and page 7 for a page size 10 should have entries 1 through 10 and 61 through 70 respectively. Consider only the division specified by **`league`**
- Case 2: If the page parameter (**page**) is not defined
    - Return all match entries with all the above return parameters. Consider only the division specified by **`league`**

TASK 5

Route 4: `/players`

Description: Returns an array of selected player attributes sorted by their names in alphabetical order.
Route Parameter(s): *None*
Query Parameter(s): **`page`** (int)*, **`pagesize`** (int)* (default: `10`)
Route Handler: **`all_players(req, res)`**
Return Type: JSON
Return Parameters: {**`results`** (JSON array of { **`PlayerId`** (int), **`Name`** (string), **`Nationality`** (string), **`Rating`** (int), **`Potential`** (int), **`Club`** (string), **`Value`** (string) }) }
Expected (Output) Behaviour:

- Case 1: If the page parameter (**page**) is defined
    - Return player entries with all the above return parameters for that page number by considering the **`page`** and **`pagesize`** parameters. For example, page 1 and page 7 for a page size 10 should have entries 1 through 10 and 61 through 70 respectively.

- Case 2: If the page parameter (**page**) is not defined
  - Return all player entries with all the above return parameters

----

NOTES:

- The return types are .js types, not SQL types.
- Technically, many routes are able to also handle errors and return an error message in the response, but we have ignored that for now. When not specified, you should deal with errors in any reasonable way you like, so long as the expected (non-error-case) behaviour is the same as that specified above.
- We don't tell you what table to use and what exact attributes to select. You should look at the schema, for example, to deduce what attribute from the table most closely matches 'Rating'. It is very likely that the developers of the API spec only have a general idea of the data, so expect to spend some time thinking about how to best implement it with what you have!
- You will find Dataset Information very helpful in relating the spec to the data.
- The page and pagesize attributes are helpful for server-side pagination.

Match (Specific) Route

TASK 6

Route 5: `/match`

Description: Returns an array of information about a match, specified by id.
Route Parameter(s): *None*
Query Parameter(s): **id** (int)
Route Handler: **match(req, res)**
Return Type: JSON
Return Parameters: {**results** (JSON array of { **MatchId** (int), **Date** (string), **Time** (string), **Home** (string), **Away** (string), **HomeGoals** (int), **AwayGoals** (int), **HTHomeGoals** (int), **HTAwayGoals** (int), **ShotsHome** (int), **ShotsAway** (int), **ShotsOnTargetHome** (int), **ShotsOnTargetAway** (int), **FoulsHome** (int), **FoulsAway** (int), **CornersHome** (int), **CornersAway** (int), **YCHome** (int), **YCAway** (int), **RCHome** (int), **RCAway** (int)}) }
Expected (Output) Behaviour:

- If the **id** is found return the singleton array of all the attributes available, but if the ID is a number but is not found, return an empty array without causing an error

----

Player (Specific) Route

TASK 7

Route 6: `/player`

Description: Returns information about a player, specified by id, depending on their best position in the field

Route Parameter(s): *None*

Query Parameter(s): `id` (int)

Route Handler: `player(req, res)`

Return Type: JSON

Return Parameters (required only, see below for additional optional parameters): {`results` (JSON array of { `PlayerId` (int), `Name` (string), `Age`(string), `Photo` (string), `Nationality` (string),, `Flag` (string), `Rating` (int), `Potential` (int), `Club` (string), `ClubLogo` (string) , `Value` (string), `Wage` (string), `InternationalReputation` (int), `Skill` (int), `JerseyNumber` (int), `ContractValidUntil` (int), `Height` (string), `Weight` (string), `BestPosition` (string), `BestOverallRating` (int), `ReleaseClause` (string) }) }

Expected (Output) Behaviour:

- If the `id` is found return the singleton array of all the attributes available, but if the ID is a number but is not found, return an empty array without causing an error
- Additional return parameters will vary depending on the players' `BestPosition`.
  - If the player's `BestPosition` is 'GK' you should return (in addition to the above required return parameters) the 6 goalkeeper specific rating attributes. They begin with the letters 'GK'. These attributes are { `GKPenalties` (int), `GKDiving` (int), `GKHandling` (int), `GKKicking` (int), `GKPositioning` (string), `GKReflexes` (int) }
  - For all other players, return the 6 'neutral' player rating attributes (in addition to the above required return parameters). These are: { `NPassing` (int), `NBallControl` (int), `NAdjustedAgility` (int), `NStamina` (int), `NStrength` (string), `NPositioning` (int) }
- Values like `ReleaseClause` might be NULL for some entries - return them as is.

Search Routes

TASK 8

Route 7: `/search/matches`

Description: Returns an array of selected attributes for matches that match the search query
Route Parameter(s): *None*
Query Parameter(s): `Home` (string)*, `Away` (string)*, `page` (int)*, `pagesize` (int)* (default: `10`)
Route Handler: `search_matches(req, res)`
Return Type: JSON
Return Parameters: {`results` (JSON array of { `MatchId` (int), `Date` (string), `Time` (string) `Home` (string), `Away` (string), `HomeGoals` (int), `AwayGoals` (int)}) }
Expected (Output) Behaviour:

- Return an array with all matches that match the constraints. If no match satisfies the constraints, return an empty array without causing an error
- The expected match behaviour for string-matching is the same as that of the LIKE function in MySQL
- The behaviour of the the route with regard to `page` and, `pagesize` is the same as that for routes 3 and 4
- Alphabetically sort the results by (`Home` , `Away`) attribute (i.e., sorted by home team first then the away team - both in alphabetical order)

TASK 9

Route 8: `/search/players`

Description: Returns an array of selected attributes for players that match the search query
Route Parameter(s): *None*
Query Parameter(s): `Name` (string)*, `Nationality` (string)*, `Club` (string)*, `RatingLow` (int)* (default: `0`), `RatingHigh` (int)* (default: `100`), `PotentialLow` (int)* (default: `0`), `PotentialHigh` (int)* (default: `100`), `page` (int)*, `pagesize` (int)* (default: `10`)
Route Handler: `search_players(req, res)`
Return Type: JSON
Return Parameters: {`results` (JSON array of { `PlayerId` (int), `Name` (string), `Nationality` (string), `Rating` (int), `Potential` (int), `Club` (string), `Value` (string) }) }
Expected (Output) Behaviour:

- Return an array with all players that match the constraints. If no player satisfies the constraints, return an empty array without causing an error
- For string-matching, the expected match behaviour is the same as that of the LIKE function in MySQL
- The behaviour of the the route with regard to `page` and, `pagesize` is the same as that for routes 3 and 4
- xHigh and xLow are the upper and lower bound filters for an attribute x. Entries that match the ends of the bounds should be included in the match. For example, if RatingLow

were 1 and Rating Higher were 5, then all players whose OverallRating was >=1 and <=5 would be included.

- Alphabetically sort the results by the player name (**`Name`**) attribute

----

Testing

We provide test cases that use JEST and Supertest for testing your API. Specifically, these tests will run your application, querying specific routes and checking if the response object is as expected. While these tests are made to help you check for your API's correctness in implementing the spec, they are very basic.

In addition to running the tests using **`npm test`** as described in Task 2, we encourage you to take a look at *tests.js* in the __tests__ directory to both understand how the provided tests work and to potentially write your own tests for further checking. We won't be grading your tests as part of this assignment, but developing tests will help you understand and adhere to the specifications much better while providing you with a solid background for the term project!

Submission and Grading

Once you are convinced that you have implemented the specifications and are (ideally) passing all provided test cases (worth 20 points), submit ONLY routes.js to the Homework 2-MS1 submission on Gradescope. Once the submission deadline has passed, we will grade your submissions manually, for an additional 20 points, assessing the correctness of your implementation further.