CSE 331: Software Design & Engineering Winter 2024

Homework 1

Due: Wednesday, January 15th, 11pm

Ensure you've completed the course Software Set-up, then check out the starter code for this assignment:

```
git clone https://gitlab.cs.washington.edu/cse331-25wi/materials/hw1-parsing.git
```

Navigate to the `hw1-parsing` directory with `cd hw1-parsing` and run `npm install --no-audit`.

As described in lecture, these first few assignments emphasize the **debugging process** which helps us motivate the reasoning skills we will teach for the remainder of the quarter. You will log your debugging and submit it as a PDF. We expect that you spend time understanding your bugs and carefully detailing the steps you took to do so.

Inevitably, you will spend time debugging as a computer scientist, and the problems you debug will only get harder in and after this class! So we strongly encourage that you take advantage of this opportunity to strengthen those skills and set yourself up for success in the future by earnestly engaging in the challenges of this assignment.

Campus Map App Implementation

In this assignment, you will be writing and debugging server side functions that will be used by a (provided) client side app that displays a map of the UW campus and allows users to search and select campus buildings and interact with the schedules of their friends.

You should attempt to implement *all* the server side functions described below before you try to run your app. Once you have attempted to implement all the functions, you can run the app and log your debugging process. **We recommend that you aim to finish your initial implementation by Monday afternoon (Jan 13), and leave the rest of the time for debugging.** (See Task 1 for more details about why this is important and the whole debugging process.)

Parsing

Implement the function `parseBuildings` in `src/routes.js`. This function is passed in a list of lines of text (from the file `data/campus_buildings.csv`), each of which is a string of the form:

```
MOR,Moore Hall,2317.1749,1859.502
```

with commas separating the four provided pieces of information about the building.

Parse each line into a record containing fields called `shortName`, `longName`, `x`, and `y`, corresponding to the four parts above. The two names should be strings, while `x` and `y` should be numbers. Each of these records should be added to the `buildings` array.

Then, implement the function `parseSchedules` in `src/routes.js`. This function is passed in a list of lines of text (from the file `data/campus_schedules.csv`), each of which is a string of the form:

```
James,14:30,CSE2
```

with commas separating the name of a friend (from the 331 staff!), the time, and the short name building for a class in their schedule. Parse each line and store it in data structure(s) of your choosing.

See page 8 for descriptions of functions that may be helpful for parsing. For all tasks, you should only use functions and methods listed there.

Building Routes

Implement the function `findByName` in `routes.js`. This function should find buildings (from the Parsing step above) whose long names contain the text provided in the query parameter called `text`. The "`query`" field of the request object, `req`, is a record containing each of the query parameters as a field.

Your code should send back (via the response object, `res`) a record containing a single field called "`results`" that contains an array of the buildings found. Each building should be a record as above.

Your solution must follow these additional rules:

- Return at most 3 results. If more than 3 buildings contain the provided text in their long names, then return the *first* three buildings from the original list.
- Your text matching should be case-*insensitive*. I.e., searching for "engineering" should find "Paul G. Allen Center for Computer Science & Engineering" even though the case is different.

Implement the function `findBuildingsByPosition` in `routes.js`. This function should find the **three** closest buildings to the (x, y) position provided in the query parameters `x` and `y`.

Your code should send back the results in the same format as `findByName`.

Schedule Routes

Implement the function `findByFriend` in `routes.js`. This function should find *all* scheduled classes (from the Parsing step above) for the person provided in the query parameter called `text`.

Like `findByName`, your text matching for friend's names should be case-*insensitive*, but unlike `findByName`, it is okay if it only works for full name searches (e.g. if `text` is "Ja", that should produce *no* results, rather than results for James and Jaela).

Your code should send back the results as an array of records, each containing the fields `friend`, `time`, `shortName` (the details of a scheduled class), and `x` and `y` (the coordinates of the building with `shortName`).

Implement the function `findByTime` in `routes.js`. This function should find the scheduled classes for all friends at the time provided in the query parameter called `text`.

It is okay if your implementation does not do any verification that the time is correctly formatted (HH:MM), though it can if you want.

Your code should send back the results in the same format as `findByFriend`.

Implement the function `findFriendsByPosition` in `routes.js`. This function should find the **three** `unique` friends that, at any time, will be closest to the (x, y) position provided in the query parameters `x` and `y`.

Your code should send back the results in the same format as `findByFriend`. Your results array should have exactly 3 class entries.

Task 1 – Log and Pony Show [100 pts]

In this task, you will try out your solutions to the functions you implemented by running the front end app that uses them. We ask that you fully implement the functions above before you run your app, so you can more carefully **track** and **document** your time spent debugging.

For each bug, you must also provide the following information:

- What **failure**, (incorrect) app behavior, did you see that told you there was a bug?
- Which **experiments** did you perform to try to locate the defect? (Checking the network tab, scanning for typos, `console.log`s, etc.)
- What the **defect** was that caused that bug (if you ever found it)?
- How many **minutes** did you spend on the bug after noticing the failure?
- Would a (Java) **type checker** have spotted the bug?

  Note that the type checker not only finds cases where data of the wrong type is used (e.g., a number where a string is expected) but also when the names of fields or functions are misspelled.

We have provided a debugging log website for you to use to record this information. We recommend that you have the log open while you test your app, so you remember to write all your bugs down as you find them. Remember, even a simple typo is a bug, and you should log it!

Once you have finished debugging your app, you will select 3 log entries to turn in. Assuming you have > 3 entries, you should select entries for bugs that are "interesting" (perhaps your most challenging/time-consuming bugs, bugs you can describe in detail, bugs that helped reinforce a concept we've talked about in class). We will not grade you on how "interesting" your bug selections are, but this will help you have content to write a compelling description.

Your log should capture all the necessary context around each bug. The audience for this log is your TAs, who are familiar with the starter code, app, and requirements, but are not familiar with your implementation choices. We want to be able to follow your experience and be convinced that you spent your time wisely on effective debugging steps (Experiments).

We ask that you debug your implementation for 4 hours or until you find 3 bugs to describe in your debugging log, whichever comes second. This will ensure that you get solid debugging practice but avoid having you agonize over debugging for too long (after all, we haven't shown you all the super great tools to help you write code to avoid debugging!). It is okay if you are unable to *fix* all bugs that you find, sometimes it's best to move on to something else when you've been stuck on a bug for a while, but you should still describe your process of trying to find the defect and get as close as you can.

If you think your implementation is correct, and you have not yet found 3 bugs... we still want you to have the whole debugging experience, so we ask that you send us your implementation, and we will return it to you as soon as possible with new bugs for you to debug. To send us your work, you should upload `routes.js` containing your completed functions to Google Drive, configure the share settings so we can access it by the share link, and email that link to the cse331-staff@cs.washington.edu.

If you do not find 3 bugs and need to send us your implementation, you **MUST** send it to us by **Monday, Jan 13th at 7pm**. Besides good time management, this is the reason why we highly recommend that all students try to implement their functions by Monday. However, we know bugs are extremely common even for experienced programmers, so we expect you'll find your own.

To try your solution, open the file `public/index.html` and add the following line just before `</body>`:

```
<script src="ui.js"></script>
```

Then, run the command `npm run start` in the terminal within the `hw1-parsing` directory to start the app. Open `http://localhost:8080` in Chrome to see the running campus map app.

If your code does not compile, fix the issues in the terminal output first before you start logging bugs. Bugs start with a failure in the app, meaning that **compilation issues are not bugs!** Also, you should only make debugging changes in `routes.js`, not in any of the provided starter code.

Selecting "buildings named", typing in the input box, and clicking "Go" should perform a "find by name" search. Similarly, searching with "schedule of" and "classes at" will perform "find by friend" and "find by time" searches, respectively. Clicking anywhere in the map should perform a "find by position" search; either finding the closest buildings or friends depending on the selected radio button.

In all cases, the results you returned should be shown both on the map and in a list below the input areas. If they are not, then there is a bug. Happy hunting!

The following is a set of examples of app inputs and the expected outputs to try with your app. You should definitely experiment with inputs beyond these.

- Select "buildings named" from the dropdown and type "phY" in the input area and "Go":

- With the "buildings" toggle selected, click the Women's Softball Field on the map:

- Select "schedule of" from the dropdown and type "Ali" in the input box and "Go":

- Select "classes at" from the dropdown and type "11:00" in the input box and "Go":

- With the "friends" toggle selected, click the "Chemistry" building on the map (next to Bagley):

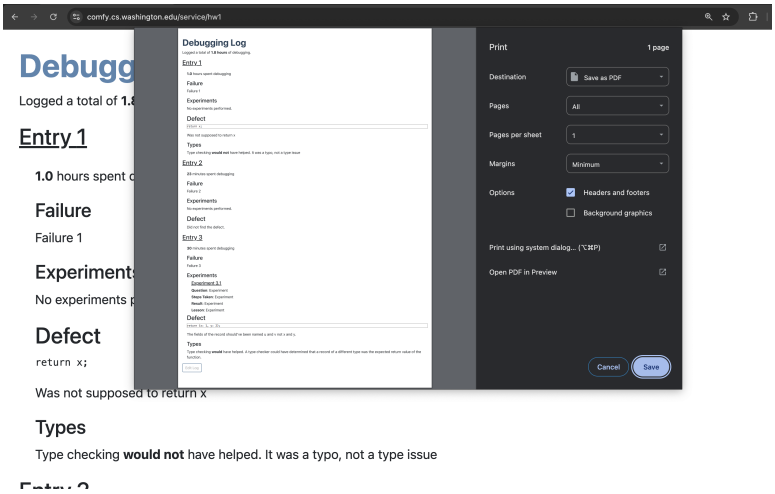
Submission

After you finish debugging:

1. Open each log entry that you want to include in your submission and select Show in "View" box.

   Show: ☑ (in "View")

2. From the main page, select "View Log" to open all of your selected log entries.
3. Select File > Print or ctrl+P/Command+P to open the print dialog.
4. Set the print destination as "Save as PDF" and "Save"



5. Submit your downloaded log to the "HW1" assignment on Gradescope.

Useful Library Functions & Methods

This page contains library functions (and methods) you can use in your solution. You will also need to use the functions shown in lecture and section for sending server responses.

Please restrict yourself to only those listed here. Specifically note that the functions from the `Math` library are **not** allowed, but mathematical operations (+ * - / **) are.

You can read more about these allowed functions (as well as many other library functions) by searching in the Mozilla Developer Network documentation on the web.

Global Functions

`parseFloat` Takes a string as an argument and returns the floating point value it describes. For example, a call to `parseFloat("3.14")` would return the floating point value 3.14. This returns `NaN` (not a number) if the string does not represent any valid number.

`isNaN` Returns true if the value passed is `NaN`, otherwise returns false.

Fields and Methods of `Array`

`length` The length field stores the length of the array, e.g., `[1, 2, 3].length` is 3.

`push` Adds the value passed in as an argument to the end of the array. E.g., if `A = [1, 2]`, then after calling `A.push(3)`, the value of `A` would be `[1, 2, 3]`.

`pop` Removes the last element from the end of the array and returns it. E.g., if `A = [1, 2]`, then after calling `A.pop()`, 2 would be returned, and the value of `A` would be `[1]`.

`slice` Called with no arguments, this returns a (shallow) copy of the array. Returns a copy of a portion of the array when called with arguments selecting the `start` (and optionally `end`) index of the portion.

`sort` Reorders the elements of the array into ascending order. For primitive types, this orders them according to the built-in "<=" operator for that type. For other types, you must pass in an argument, which is a function taking two arguments `(a, b)` and returning a negative value if $a < b$, a positive value if $a > b$, and zero if $a = b$. (For numbers, just `(a, b) => a - b` works!)

Methods of `String`

`indexOf` Takes a character (i.e., a length-1 string) as an argument and returns the first index where that character appears in the string or -1 if the character is nowhere in the string.

`includes` Returns true if the string contains the argument (another string) somewhere within it. E.g., `"a quick brown fox".includes("bro")` returns true.

`split` Takes a character (i.e., a length-1 string) as an argument and returns an array containing each of the contiguous pieces of the string without that character. For example, `"the quick brown fox".split(" ")` returns `["the", "quick", "brown", "fox"]`.

`toLowerCase` Returns the same string but with all upper-case characters replaced by their lower-case versions (all other characters are unchanged).

Fields and Methods of `Map`

`set` Accepts a key and value as arguments and adds the pair to the map, replacing the current value if a pair with the given key already exists.

`has` Returns true if the key passed in as an argument exists in a (key, value) pair in the map.

`get` Returns the value paired with the key passed in as an argument if that key exists, otherwise undefined.

`keys` Returns an iterator over all keys in the Map.

`values` Returns an iterator over all values in the Map.