CSE 154: Web Programming Summer 2019

Practice Exam 2B

Name:

UWNet ID: @uw.edu

TA (or section):

Rules:

- You have 60 minutes to complete this exam.
- You will receive a deduction if you keep working after the instructor calls for papers.
- You may not use any electronic or computing devices, including calculators, cell phones, smartwatches, and music players.
- Unless otherwise indicated, your code will be graded on proper behavior/output, not on style.
- Do not abbreviate code, such as writing ditto marks (”) or dot-dot-dot marks (...). You may not use JavaScript frameworks such as jQuery or Prototype when solving problems.
- If you enter the room, you must turn in an exam and will not be permitted to leave without doing so.
- You must show your Student ID to a TA or instructor for your submitted exam to be accepted.

Question	**Score**	**Possible**
Short Answer		16
JS with AJAX		10
Node.js/Express Web Service		14
SQL		10
XC		

1. Short Answers

1. Data Storage Trade-offs.

a. What is one advantage of using a SQL database over text files to store data?

b. From the client perspective, what is an advantage of working with a JSON response over one in plain text format?

2. Node Modules. Why do we install most Node modules locally, rather than globally (with the **-g** flag) ?

3. Server-side Error-Handling.

a. Provide a specific example where it would be more appropriate to return a 400 error instead of a 500 error in a web service:

b. Provide a specific example where it would be more appropriate to return a 500 error instead of a 400 error in a web service:

4. Asynchronous Program Execution. Consider the following JS program:

```
“use strict”;
let t1 = null;
let t2 = null;
let doggoCount = 0;

t1 = setInterval(doggo, 300);
t2 = setTimeout(dubs, 800);

function doggo() {
  doggoCount += 1;
  console.log(doggoCount + " doggo");
}

function dubs() {
  console.log("DUBS!");
  t1 = null;
  clearInterval(t1);
  t2 = setTimeout(dubs, 800);
}
```

Circle which of the following options would be correct as the first 8 lines of console output when the program is ran (fewer than 8 lines indicate no more console output is possible until the program is restarted).

a	b	c	d	e
1 doggo 2 doggo DUBS! 3 doggo 4 doggo DUBS! 5 doggo 6 doggo	1 doggo 2 doggo 3 doggo DUBS!	1 doggo 2 doggo DUBS! 1 doggo 2 doggo DUBS! 1 doggo 2 doggo	1 doggo 2 doggo DUBS! 3 doggo 4 doggo 5 doggo DUBS! 6 doggo	1 doggo 2 doggo DUBS! DUBS! DUBS! DUBS! DUBS! DUBS!

2. Node.js Web Service: Club CSE 154!

The CSE 154 TAs are developing a brand new game called Club CSE 154. It is filled with many notable references made throughout the Spring 2019 quarter. The team has compiled a bunch of folders filled with images and descriptions of each notable reference. In this question, you will implement the Node web service **app.js** that will eventually allow us to build a simple character showcase webpage with client-side JS on the next problem.

Directory Structure

Your web service is located in the same level as the **public** directory, which contains subdirectories corresponding to each of the character names. Each subdirectory contains 3 files:

- **avatar.png**, an image of the character.
- **info.txt**, a document with three lines of information about the character:
  - **name**
  - **series**
  - **description**
- An **appearances.txt** file which contains a line for each unique appearance of the character in CSE154 19sp. For example, a file with 4 lines would correspond to 4 appearances during the quarter. This file is guaranteed to have at least one line.

We have included the newline ("\n") character in bold where relevant.

```
club154/
└── app.js
   package.json
   public/
    └── club154.html
      club154.css
      club154.js
      charlist/
       └── corrin/
          └── appearances.txt
            avatar.png
            info.txt
          debugduck/
           └── appearances.txt
             avatar.png
             info.txt
          mowgli/
           └── appearances.txt
             avatar.png
             info.txt
         ...
```

`info.txt` Content Structure:

<name info>**\n**
<series info>**\n**
<description>**\n**

Example file Contents:

(charlist/debugduck/info.txt)

Debug Duck**\n**
Programming Aid**\n**
A guiding light through your code storms.**\n**

`appearances.txt` Content Structure:

<first appearance>**\n**
<second appearance>**\n**
…
<last appearance>**\n**

Example File Contents:

(charlist/debugduck/appearances.txt)

Course website homepage illustration**\n**
4/19 Lecture's Debug Duck giveaway**\n**

API Documentation

The **app.js** API supports two **GET** requests. For both, if any file- or directory- processing error occurs, send a 500 error using the provided `SERVER_ERROR_MSG` constant, "Something went wrong on the server, please try again later.". Specific details for each GET request (including specific 400 errors) are described below:

Query 1: Get all Characters

Request Format: **`/characters/all`**
Request Type: **`GET`**
Returned Data Format: **`plain text`**
Description: Outputs a plain text response with each character's directory on a new line.
Output (complete): Below is the expected output according to the data directory shown on the previous page:

```
corrin
debugduck
mowgli
...
```

Query 2: Get Data for a Single Character

Request Format: **`/characters/lookup/:charname`**
Request Type: **`GET`**
Returned Data Format: **`plain text`**
Description: Outputs a JSON response with information about a specific character, where the **`:charname`** value corresponds to a directory name that was returned from Query 1. For example, to get information about the character "Debug Duck," you would use **`debugduck`** for the **`:charname`** value.
Example Request: **`/characters/lookup/debugduck`**
Output: The general format of the JSON response is as follows:

```
{
  name: <charname>
  series: <seriesname>
  description: <description>
  appearances: [<appearance1>, <appearance2>, ...]
}
```

Each **`<string>`** above is replaced with the corresponding information unique to the character. You may assume that the response will contain valid information and that there is at least one appearance. You may also assume that if the **`:charname`** parameter is passed, it corresponds to a valid character directory on the server. *An example request and response is provided on the next page.*

Example Request: `/characters/lookup/debugduck`
Example Response:

```
{
  “name”: “Debug Duck”,
  “series”: “Programming Aid”,
  “description”: “A guiding light through your code storms.”,
  “appearances”: [
    “Course website homepage illustration”,
    “4/19 Lecture’s Debug Duck giveaway”
  ]
}
```

Error Handling: If a user attempts to lookup a character that is not present, you should return a 400 error code with the text, “Please pass in a valid character name.”

`app.js` Outline

You will implement the two endpoints as Part A and Part B, completing the rest of the following app.js (you may assume all modules are installed in the project directory). For glob and fs, you may choose to use the provided promisified functions or the callback versions in the respective modules.

For full credit, your app.js must always send the endpoint's response, may not overwrite content headers, and may not modify the response after it has been sent.

```
"use strict";
const express = require("express");
const util = require("util");
const fs = require("fs");
const glob = require("glob");
const readFile = util.promisify(fs.readFile);
const readdir = util.promisify(fs.readdir);
const globPromise = util.promisify(glob);

const app = express();

app.use(express.static("public"));

// Your solution to Part A will be graded assuming it goes here

// Your solution to Part B will be graded assuming it goes here

const PORT = process.env.PORT || 8000;
app.listen(PORT);
```

Part A: Get all Characters

Implement the **`/character/all`** GET endpoint to replace the // Part A comment in the app.js outline. This endpoint should return a string concatenating each directory name in the **`public/`** directory. Each directory name should be appended with a newline ("\n") character at the end. For example, in our sample directory, this request should send the following response (making sure to handle possible 500 errors as described in the API documentation):

```
corrin\n
debugduck\n
mowgli\n
...
```

```
// Your solution to Part A goes below:
```

Part B: Building JSON for Query 2

Implement the GET endpoint **`/characters/lookup/:charname`** to replace the // Part B comment in the app.js outline. For this endpoint, you should use **:charname** as the character directory name, and send a **JSON** response having the following form:

```
{
  “name”: <character name>,
  “series”: <character series>,
  “description”: <character description>,
  “appearances”: <array of appearances>
}
```

continuing the character information described in the API documentation for Query 2. Make sure to handle possible 500 errors as described in the API documentation.

```
// Your solution to Part B goes below:
```

4. Gotta Fetch 'em All!

Now that the Club CSE 154 API is finished (and you may assume it is implemented correctly), the CSE 154 TAs are looking for someone to implement the "Choose your Character" webpage to showcase each featured CSE 154 character in a "character spotlight". You are hired to do just that. Welcome to the team!

The TAs have designed two sample screenshots and a specification that you are to work from, detailed below.

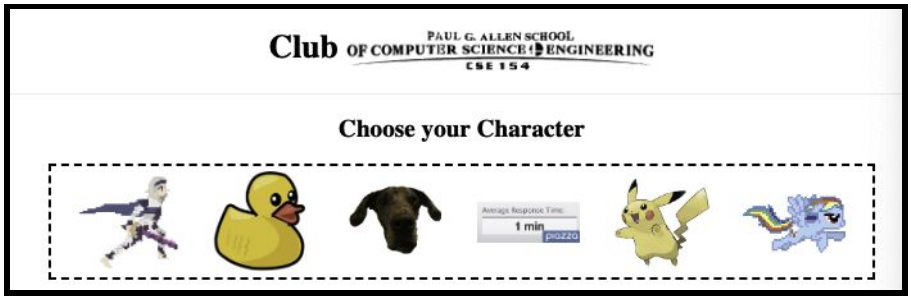


Figure 1: When the page is loaded (after the character images have been added to the page)

Figure 2: After a user selects a character image (e.g. the dog Mowgli)

Behavior Details

For this page, your JS will fetch data from the endpoints configured in the previous problem. All files necessary should be stored in the **public** directory alongside your HTML/CSS/JS files. Be sure to take this into account when accessing images directly.

The body of the provided **club154.html** file is provided below for reference:

```
<body>
  <h1>Club <img src="logo.png" alt="CSE 154"></h1>
  <hr>
  <h2>Choose your Character</h2>
  <div id="playground"></div>

  <section id="spotlight" class="hidden">
    <h2>Club CSE 154 Spotlight: <span id="name"></span></h2>
    <p><span id="series"></span> Series</p>
    <p id="description"></p>
    <h3>Guest Appearances</h3>
    <ul id="appearances"></ul>
  </section>
  <p id="error" class="hidden">Club CSE 154 is currently out of service :(</p>
</body>
```

Initial View

When the page loads:

- A request should be made to **`/character/all`**. Recall that this request returns a plain text response listing all character directory names, each as a single line.
- For each line returned in a successful request, an image element should be added to the **`#playground`** with the path **`<chardir>/avatar.png`**, replacing **`<chardir>`** with the line text (e.g. **debugduck**). The resulting page will appear as seen in Figure 1.
- Each image should be given an ID corresponding to the directory name for use in Query 2 requests later.

Character Spotlight

When one of the added character images is clicked:

- Data about that character should be requested from the API (using Query 2, **`/characters/lookup/:name`**). The response JSON should be used to populate the **`#spotlight`** view:
  - **`#name`** should be populated with the returned name.
  - **`#series`** should be populated with the returned series name.
  - **`#description`** should be populated with the returned character description.
  - **`#appearances`** should be populated with a list of appearances, with one list item for each appearance in the returned appearances array.
- The clicked image should be given the **`.selected`** class (assume provided in CSS), which includes the styling for the blue border. Figure 2 shows an example of what the page would look like after Mowgli's (the dog) image is clicked.
- **`#spotlight`** should be made visible if not visible already. The **`.hidden`** class, which you may assume is provided in the CSS to hide/show elements, is described in more detail in the **Error Handling** section and will have the same behavior as in HW3/HW4.
- At most one character image should have the **`.selected`** class at any given time. If an image with **`.selected`** is clicked, nothing should change on the page (until a different image is clicked).

- Note that **`#appearances`** should only ever contain appearances for the current spotlight character.

Error Handling

If an error occurs in any fetch request:

- The **`#error`** element should be displayed and the **`#spotlight`** view should be hidden.
- No element should have the **`.selected`** class.
- If an image is clicked again and the request is successful, **`#error`** should be hidden and **`#spotlight`** should display again, resuming the behavior as described in the **Character Spotlight** section above.

Write your JS solution below. You may assume that the **`checkStatus`** function and the aliases **`id(idName)`**, **`qs(el)`**, and **`qsa(sel)`** are defined for you and included as appropriate. An additional page is available if necessary.

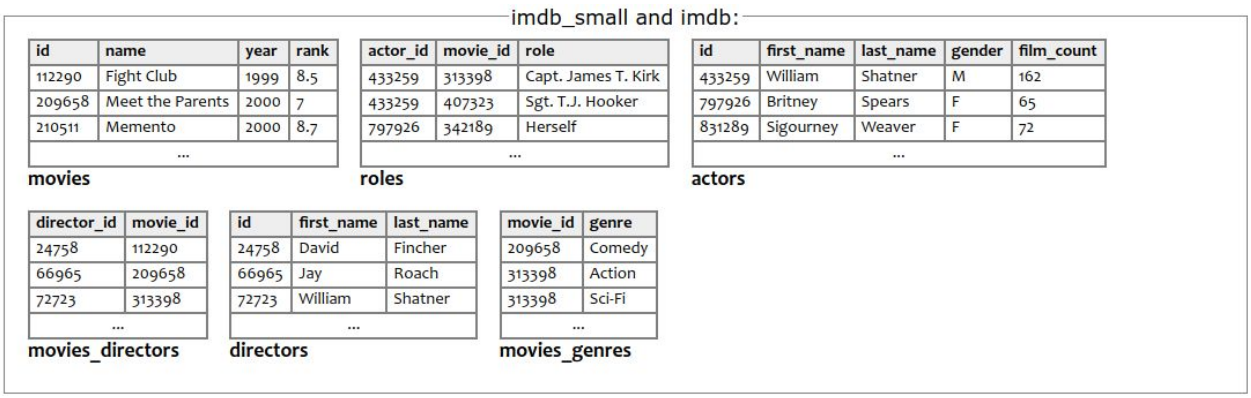
```
"use strict";
(function() {
  const API_URL = "/characters";

  window.addEventListener("load", init);
```

```
})();
```

4. SQL Queries

Recall the IMDB (Movies) database from the CSE 154 Query Tester:



1. Write a SQL query to list the first and last names of all the female actors that have a first name that starts or ends with “W”.

Expected results (5 rows returned total):

first_name	last_name
Wendy Lee	Avon
Wilma Jeanne	Cummins
Wynter	Kullman
W. Lauren	Sanchez
Meadow	Williams

2. Write a SQL query to list all columns out of the directors table for all directors that have directed a movie with a rank of 8 or higher, order by last name of director in alphabetical order. Return one row per unique director.

Expected results (20 rows returned total):

id	first_name	last_name
429	Andrew	Adamson
9247	Zach	Braff
11652	James (I)	Cameron
...	...	...