Note that this practice exam is slightly longer (~5-10 minutes) than you should expect to provide more practice when studying.

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.

<table>
<thead>
<tr>
<th>Question</th>
<th>Score</th>
<th>Possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Answer</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>JS with AJAX</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Node.js/Express Web Service</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>SQL</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>XC</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>50</td>
</tr>
</tbody>
</table>
1. **Short Answers**

1. **Validation Methods.** What is one example of validating user input on the client-side?

2. **GET vs. POST.** Provide and justify a specific example when it is more appropriate to use a POST request instead of a GET request (2-3 sentences).

3. **Regex.** For each of the two regular expressions, circle all the string(s) below that match it:
   i. `/^[A-Z]+4\d\d$/`
   - CSE431
   - CSE154
   - http400
   - ABC4dd
   - 404

   ii. `/^pup.*y.jpg$/`
   - puppy.jpg
   - `^puppypuppy.jpg$
   - pupy.jpg
   - puppyparty.jpg
   - puppykitty.JPG

Regex reference:

<table>
<thead>
<tr>
<th>[abc]</th>
<th>A single character of: a, b, or c</th>
</tr>
</thead>
<tbody>
<tr>
<td>^[abc]</td>
<td>Any single character except: a, b, or c</td>
</tr>
<tr>
<td><code>[a-zA-Z]</code></td>
<td>Any single character in the range a-z or A-Z</td>
</tr>
<tr>
<td>^</td>
<td>Start of line</td>
</tr>
<tr>
<td>$</td>
<td>End of line</td>
</tr>
<tr>
<td>\A</td>
<td>Start of string</td>
</tr>
<tr>
<td>\Z</td>
<td>End of string</td>
</tr>
</tbody>
</table>

- \s: Any whitespace character
- \S: Any non-whitespace character
- \d: Any digit
- \D: Any non-digit
- \w: Any word character (letter, number, underscore)
- \W: Any non-word character
- \b: Any word boundary
- (...): Capture everything enclosed
- (a|b): a or b
- a?: Zero or one of a
- a*: Zero or more of a
- a+: One or more of a
- a{3}: Exactly 3 of a
- a{3,}: 3 or more of a
- a{3,6}: Between 3 and 6 of a

| options:   | i | case insensitive | m | make dot match newlines | x | ignore whitespace in regex | o | perform #\{\} substitutions only once |
4. **fs, glob, and path modules.** Suppose a directory has the following structure (with package.json/node_modules included as appropriate):

```javascript
app.js
mydir/
    images/
        puppy1.jpg
        puppy1.png
        puppy2.gif
        puppy-facts.txt
        puppy-haz-pizza.jpg
```

Consider the following statement written in app.js, **assuming readdir and globPromise are promisified versions of fs.readdir and glob**:

```javascript
let result = STATEMENT;
```

For each statement replacing STATEMENT, what would be the value of result? Use [] for any Arrays and "" for Strings.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Value of result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>await readdir(&quot;mydir&quot;);</code></td>
<td></td>
</tr>
<tr>
<td><code>await readdir(&quot;mydir/images&quot;);</code></td>
<td></td>
</tr>
<tr>
<td><code>await globPromise(&quot;mydir/puppy-facts.txt&quot;);</code></td>
<td></td>
</tr>
<tr>
<td><code>await globPromise(&quot;mydir/**&quot;);</code></td>
<td></td>
</tr>
<tr>
<td><code>await globPromises(&quot;mydir/puppy*&quot;);</code></td>
<td></td>
</tr>
<tr>
<td><code>path.extname(&quot;mydir/images/puppy1.jpg&quot;);</code></td>
<td></td>
</tr>
<tr>
<td><code>path.basename(&quot;mydir/images/puppy1.jpg&quot;);</code></td>
<td></td>
</tr>
</tbody>
</table>

5. **Data Storage Methods.** For each data storage method, mark "X" in each column for the location it can be accessed during an HTTP request/response in Node.js/Express. Note that some methods may have X in both Client and Server columns.

<table>
<thead>
<tr>
<th>Data Storage Method</th>
<th>Client (Browser)</th>
<th>Server (Node.js/Express)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The website’s DOM.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text Files</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL databases</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cookies</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6. **Node/mysql Connection.** Why is it important to use ? placeholders in `db.query` when using promise-mysql in a Node.js app? (2-3 sentences).
2. (JS with Fetch): Plan-It! Fetching You Meals a Day at a Time

In this question, you will write JavaScript to implement a small Meal Planner web page called Plan-It!. For simplicity, we will consider a "full day meal plan" as a breakfast, lunch, and dinner (the 3 standard meal types). Below are two screenshots of the Plan-It! page:

### Plan-It!

**Click the button to generate a random 3-meal menu plan!**

**Fetch Random Menu**

**Initial View**

### Plan-It!

**Click the button to generate a random 3-meal menu plan!**

**Fetch Random Menu**

**Cheerios Cereal (Breakfast)**

1 cup of Original Cheerios with 1/2 cup of soymilk.

Food Groups:
- Dairy
- Grains

**Froot Loops Cereal (Lunch)**

A baggy of rainbow Froot Loops - a classic college student lunch.

Food Groups:
- Grains

**Pinkie Pies (Dinner)**

Two pies made with love and filled full of flavors. Blueberry, strawberry, you name it, you have it!

Food Groups:
- Grains
- Fruit
- Other

**After Fetching Random Full-Day Menu**

### Directory Structure

You will fetch from a Node.js API you implement in Problem 2. Assume your all Problem 2 and Problem 3 files are inside a `planit` directory as follows, where `app.js` correctly serves the static `public/` directory (similar to the HW4 and the Final Project required directory structures).

```
planit/
  └─ public/
      └─ planit.html
          planit.js
          styles.css
  app.js
  package.json
  data/
  // meal directories specific to Problem 3, irrelevant in Problem 2
```
Implementation Requirements

In planit.js, you will fetch using **only one** of the two GET endpoints you will implement in app.js for Problem 3.

Clicking the #day-btn should make a fetch request to the API's /dayplan GET endpoint to fetch a random full day meal plan (one option for each of breakfast, lunch, and dinner). The response JSON format will be in the following format (example):

```json
{
  "breakfast": {
    "name": "Blueberry Oatmeal",
    "description": "One cup of hot oatmeal with 1/4 cup of fresh blueberries."
    "food-groups": ["Grains", "Fruit"]
  },
  "lunch": {
    "name": "Froot Loops",
    "description": "A baggy of rainbow Froot Loops - a classic college student lunch."
    "food-groups": ["Grains"]
  },
  "dinner": {
    "name": "Spaghetti with Tomato Sauce",
    "description": "Two pies made with love and filled full of flavors. Blueberry, strawberry, you name it, you have it!"
    "food-groups": ["Grains", "Fruit", "Other"]
  }
}
```

Upon success, remove the .hidden class from the #day-results (you will not ever need to add it back) and use the response JSON to populate each of the #breakfast, #lunch, and #dinner aside elements on the page such that:

- The .name element is populated with the corresponding meal name.
- The .description element is populated with the corresponding meal description.
- The .food-groups element is populated with a list of the food groups for that item (one or more food groups may be in the "food-groups" array in the JSON).

A client should be able to click the #day-btn multiple times to get a new randomly-generated 3-meal menu (previous menu results should be replaced as a result).

Note: You do not need to know any CSS used by the page other than the .hidden class which should be added/removed as previously specified.
Click the button to generate a random 3-meal menu plan!
<button id="day-btn">Fetch Random Menu</button>

<section id="day-results" class="hidden">
<article id="breakfast">
<h2><span class="name"></span> (Breakfast)</h2>
<p class="description"></p>
<p>Food Groups:</p>
<ul class="food-groups"></ul>
</article>
<article id="lunch">
<h2><span class="name"></span> (Lunch)</h2>
<p class="description"></p>
<p>Food Groups:</p>
<ul class="food-groups"></ul>
</article>
<article id="dinner">
<h2><span class="name"></span> (Dinner)</h2>
<p class="description"></p>
<p>Food Groups:</p>
<ul class="food-groups"></ul>
</article>
</section>
Write your client-side JS solution below. You may assume that the checkStatus(resp) function and the aliases id(idName), qs(el), qsa(sel), and gen(tagName) are defined for you and are included as appropriate.

"use strict";
(function() {

});()
3. (Node.js Web Service) Serving Up Some Meal Ideas

In this question, you will implement the Plan-It API (app.js) you were asked to use in Problem 2.

Directory Structure and File Format Details

app.js will be located in the data/ subdirectories corresponding to each of the standard meals in a day (breakfast, lunch, and dinner). You may assume these are the only subdirectories in data.

<table>
<thead>
<tr>
<th>Project Directory Structure</th>
<th>Example data/ Directory Contents:</th>
</tr>
</thead>
<tbody>
<tr>
<td>app.js</td>
<td>breakfast/</td>
</tr>
<tr>
<td>public/</td>
<td>banana.txt</td>
</tr>
<tr>
<td>planit.html</td>
<td>blueberry-oatmeal.txt</td>
</tr>
<tr>
<td>planit.js</td>
<td>...</td>
</tr>
<tr>
<td>styles.css</td>
<td>lunch/</td>
</tr>
<tr>
<td>data/</td>
<td>froot-loops.txt</td>
</tr>
<tr>
<td>breakfast/</td>
<td>spinach-salad.txt</td>
</tr>
<tr>
<td>&lt;foodoption&gt;.txt</td>
<td>...</td>
</tr>
<tr>
<td>&lt;foodoption&gt;.txt</td>
<td>dinner/</td>
</tr>
<tr>
<td>...</td>
<td>brown-rice-with-veggies.txt</td>
</tr>
<tr>
<td>&lt;foodoption&gt;.txt</td>
<td>pinkie-pies</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Inside of data subdirectory (which you may assume is non-empty), are .txt files for different food options for that subdirectory meal type, each having three lines:

<table>
<thead>
<tr>
<th>.txt File Content Format:</th>
<th>Example File Contents: (blueberry-oatmeal.txt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Blueberry Oatmeal</td>
</tr>
<tr>
<td>description</td>
<td>One cup of hot oatmeal with 1/4 cup of fresh blueberries.</td>
</tr>
<tr>
<td>food groups</td>
<td>Grains Fruit</td>
</tr>
</tbody>
</table>

where name is the full name of the food option, the description is a short description, and food groups lists any major food groups associated with the food item (on a single line).

Web Service Implementation

You will implement 2 GET requests in app.js as Part A and Part B, described on the next page.
Part A: GET /meal/:type
This request should return a plain text response of all food options available in the corresponding
type directory (ignoring letter-casing, so "breakfast" is treated the same as "BREAKfast"). Each food
option should be output on a new line in the format of:

name: description

where name corresponds to the first line of the .txt file for that food item and description is the
second line. The order of lines in the output not matter. For example, a GET request to
/meal/breakfast might output the plain text:

Banana: A good source of potassium on the go.
Blueberry Oatmeal: One cup of hot oatmeal with 1/4 cup of fresh blueberries.
Cheerios Cereal: One cup of original Cheerios with 1/2 cup of soymilk.

Note that this response corresponds to three breakfast meal options in the breakfast directory, but
your request should work for any number of txt files that may be found in the directory.

Part B: GET /dayplan
This request should return a JSON response containing a random option from each meal type. One
example response may look like the following (identical to the example we gave in Problem 3):

```json
{
  "breakfast" : {
    "name" : "Blueberry Oatmeal",
    "description" : "One cup of hot oatmeal with 1/4 cup of fresh blueberries.",
    "food-groups" : ["Grains", "Fruit"]
  },
  "lunch" : {
    "name" : "Froot Loops",
    "description" : "A baggy of rainbow Froot Loops - a classic college student lunch.",
    "food-groups" : ["Grains"]
  },
  "dinner" : {
    "name" : "Spaghetti with Tomato Sauce",
    "description" : "Two pies made with love and filled full of flavors. Blueberry, strawberry, you name it, you have it!",
    "food-groups" : ["Grains", "Fruit", "Other"]
  }
}
```

Error Handling
Your web service should handle the following errors (in order of highest priority to lowest priority). If
any error occurs, only output the respective error message in plain text.

- If, when lower-cased, the :type parameter for meal/:type does not correspond to a directory
  in data/, send a 400 error with the message: "<type> not a valid meal type.",
  replacing <type> with the value of the :type path parameter (using the same letter-casing as the
  parameter passed by the client).
- 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.".

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.
You will write your solution to `app.js` below, completing the sections labeled Part A and Part B:

```javascript
"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/"));

const SERVER_ERROR_MSG = "Something went wrong on the server, please try again later.";

// Part A: Implement GET /meal/:type endpoint
```
// Part B: Implement GET /dayplan endpoint. While not required, we suggest implementing a
// getRandOption(mealType) function below your endpoint to factor out building the data
// for each of the three data/ subdirectories.

const PORT = process.env.PORT || 8000;
app.listen(PORT);
4A. (SQL table creation/insertion): Preparing the Dining Table.

After some file recovery following a fatal ENOENT error, your team has decided to move their directory structures to a SQL database. At the moment, our new Foods setup.sql contains a single table called groupings to associate foods with their respective food groups - this allows us to associate one unique food element (e.g. "Blueberry oatmeal") with potentially multiple groups (e.g. "Grains" and "Dairy").

The `CREATE TABLE` statement for this table is shown below, with an example of the corresponding table with a few rows it may eventually have:

```sql
CREATE TABLE groupings (
    group_name VARCHAR(32) PRIMARY_KEY,
    food_id INT,
    FOREIGN KEY food_id REFERENCES foods(id)
);
```

In order for this groupings table to be created, we need to first have a `CREATE TABLE` statement for the referenced foods table. Your job is to implement the foods table creation, which will represent individual food items to replace the directory structure in the data directory from Problems 2/3. Below is an example of a single row in the table corresponding to the same date for the dinner item in the example /dayplan request of Problems 2/3.

<table>
<thead>
<tr>
<th>id</th>
<th>food_name</th>
<th>description</th>
<th>meal_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Blueberry Oatmeal</td>
<td>One cup of hot oatmeal with 1/4 cup of fresh blueberries.</td>
<td>breakfast</td>
</tr>
</tbody>
</table>

Note that the `id` value should be automatically incremented each time a new record is added to this table.

1. Write the `CREATE TABLE` statement which will initialize the foods table below, requiring:
   - food_name strings having no more than 100 characters, description strings having no more than 255 characters, and meal_type strings having no more than 20 characters
   - food_name and meal_type are required, but description defaults to NULL
2. Write the INSERT statement (assuming your CREATE TABLE is correct) to insert the example row data for Blueberry Oatmeal as the first row in the foods table:

```
food id = 1
```

4B. (SQL Queries): Around the World in SQL

This question uses the World database from the CSE 154 Query Tester:

```
<table>
<thead>
<tr>
<th>code</th>
<th>name</th>
<th>continent</th>
<th>independence_year</th>
<th>population</th>
<th>gnp</th>
<th>&quot;head_of_state&quot;</th>
<th>...</th>
<th>country_id</th>
<th>language</th>
<th>official</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFG</td>
<td>Afghanistan</td>
<td>Asia</td>
<td>1978</td>
<td>22100000</td>
<td>3973.0</td>
<td>Mohammad Omar</td>
<td></td>
<td></td>
<td>Farsi</td>
<td>T</td>
<td>51.4</td>
</tr>
<tr>
<td>NLD</td>
<td>Netherlands</td>
<td>Europe</td>
<td>1581</td>
<td>15864000</td>
<td>171362.0</td>
<td>Beatrix</td>
<td></td>
<td></td>
<td>Dutch</td>
<td>T</td>
<td>95.6</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>code</th>
<th>name</th>
<th>country</th>
<th>district</th>
<th>population</th>
</tr>
</thead>
<tbody>
<tr>
<td>1235</td>
<td>New York</td>
<td>USA</td>
<td>New York</td>
<td>8008278</td>
</tr>
<tr>
<td>1236</td>
<td>Los Angeles</td>
<td>USA</td>
<td>California</td>
<td>3994820</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>country_code</th>
<th>district</th>
<th>population</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237</td>
<td>New York</td>
<td>USA</td>
<td>New York</td>
<td>8008278</td>
</tr>
<tr>
<td>1238</td>
<td>Los Angeles</td>
<td>USA</td>
<td>California</td>
<td>3994820</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>code</th>
<th>language</th>
<th>name</th>
<th>official</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFG</td>
<td>Farsi</td>
<td>Mohammad Omar</td>
<td>T</td>
<td>51.4</td>
</tr>
<tr>
<td>NLD</td>
<td>Dutch</td>
<td>Beatrix</td>
<td>T</td>
<td>95.6</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>code</th>
<th>language</th>
<th>name</th>
<th>official</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1239</td>
<td>Creole English</td>
<td>Dominica</td>
<td>T</td>
<td>100.0</td>
</tr>
<tr>
<td>1240</td>
<td>Creole English</td>
<td>Saint Kitts and Nevis</td>
<td>T</td>
<td>100.0</td>
</tr>
<tr>
<td>1241</td>
<td>Creole English</td>
<td>Antigua and Barbuda</td>
<td>T</td>
<td>95.7</td>
</tr>
<tr>
<td>1242</td>
<td>English</td>
<td>Bermuda</td>
<td>T</td>
<td>100.0</td>
</tr>
<tr>
<td>1243</td>
<td>English</td>
<td>Gibraltar</td>
<td>T</td>
<td>88.9</td>
</tr>
<tr>
<td>1244</td>
<td>Faroese</td>
<td>Faroe Islands</td>
<td>T</td>
<td>100.0</td>
</tr>
</tbody>
</table>
```

a. Write a SQL query to list the name and gnp of all countries in Asia with a gnp greater than 123456.

**Expected results (9 rows total):**

```
<table>
<thead>
<tr>
<th>name</th>
<th>gnp</th>
</tr>
</thead>
<tbody>
<tr>
<td>India</td>
<td>447114.00</td>
</tr>
<tr>
<td>Iran</td>
<td>195746.00</td>
</tr>
<tr>
<td>Japan</td>
<td>3787042.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

b. Write a SQL query to list the name of all countries having a population of fewer than 85,000 and which have languages spoken (but not necessarily official) by a percentage of at least 50% of the population. Include the language name and language percentage in your results. Order the results by language name (alphabetically increasing) breaking ties by percentage (decreasing). *Hint: The percentage column in the languages table represents the percentage of the population speaking that language in the corresponding country.*

**Expected results (16 rows total):**

```
<table>
<thead>
<tr>
<th>language</th>
<th>name</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creole English</td>
<td>Dominica</td>
<td>100.0</td>
</tr>
<tr>
<td>Creole English</td>
<td>Saint Kitts and Nevis</td>
<td>100.0</td>
</tr>
<tr>
<td>Creole English</td>
<td>Antigua and Barbuda</td>
<td>95.7</td>
</tr>
<tr>
<td>English</td>
<td>Bermuda</td>
<td>100.0</td>
</tr>
<tr>
<td>English</td>
<td>Gibraltar</td>
<td>88.9</td>
</tr>
<tr>
<td>Faroese</td>
<td>Faroe Islands</td>
<td>100.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
c. Write a SQL query to list the **city name** and **continent name** of all cities containing the word "sea" which are cities in countries having English as an official language. Order the results by city name (increasing alphabetically).

**Expected results (3 rows total):**

<table>
<thead>
<tr>
<th>name</th>
<th>continent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td>North America</td>
</tr>
<tr>
<td>Southend-on-Sea</td>
<td>Europe</td>
</tr>
<tr>
<td>Swansea</td>
<td>Europe</td>
</tr>
</tbody>
</table>

**Write your SQL query below:**
X. Extra Credit (1pt)
For this question, you can get 1 point of extra credit (demonstrating at least 1 minute’s worth of work and being appropriate in content).

Write a poem for your TA or draw a picture of them on their perfect summer vacation.