1. Short Answers

1.1. (3 pts) Concept Matching. For each item on the left **write** the letter of the matching definition on the right.

___F____ Semantic Tags

___B____ SessionStorage

___C____ Cookies

___I____ Database

___E____ Server-side

___H____ LocalStorage

___A____ Cascading Style Sheet

___G____ Callback

___D____ Separation of Concerns

1.2. (4pts) Client-side JavaScript, callbacks/timers/events

```
Countdown from 4
Countdown set
Remaining: 4
Remaining: 3
Remaining: 2
Remaining: 1
Are we there yet?
Final!
```

1.3. (2pts) Error handling.

a. When setting an error code on the server side, we usually use either 400 or 500. What is the difference between these two error codes?

Possible answers:

- 400 is request error, 500 is a server error
- One's for client error (e.g., bad parameter) and the other's for a server config error (e.g., database down).

b. We require a `checkStatus` function in our fetch call chain. What would happen if a bad response was returned, but we did **not** include `checkStatus`?

Possible answers:

- Fetch could fail silently and we wouldn't have an opportunity to present an error message to users.
- .catch would never be called because nothing throws an error. This prevents errors from being handled correctly.
- The .then chain would attempt to parse data from a bad response, and get bad data as a result. What displays to the user is undefined.

1.4. (1pt) GET vs. POST

It's possible to simply use POST for all requests for a web service, thereby allowing larger params and enabling encryption of said params.

What is **one** reason it is more appropriate to use GET for some requests? Provide an example of an appropriate GET request.

Possible reasons:

- Users want to be able to share a URL/feature with others or bookmark a page (think: link to youtube video at a specific time)
- Convention
- It's really just another page posing as a GET request (our "normal" requests like http://localhost/index.html is actually a GET request), but we're parsing it as path parameters (e.g., /pokedex/:name")

Examples:

- https://www.youtube.com/watch?v=996ZgFRENMs&t=12s
- /bestreads/info/:name

1.5. (5pts) RegEx -- what matches?

For each of these two regular expressions, indicate which of the strings below them would match.

`/^<.[href][^\s]*>.?<.+>$/`	`/^([1-9]|[a-f])*\.0+a*$/`
1. `< href ====>A<./>` **2. `<ah><\ah>`** 3. `<wf >CSE154<\.>` **4. `< fs.read> <\fs.read>`** 5. `<a href></a>`	1. `(2i).a` 2. `1-9.0+` **3. `c.0`** 4. `1a0a` **5. `feed.0aaa`**

b. A RegEx can be a powerful tool for doing input validation. What's one reason (and why) to **not** use a regular expression?

Anything in the vein of maintainability due to their complexity would be acceptable here. Variations include:

- They're hard and so make maintainability harder
- They're brittle and can stop working when things change
- If someone else wanted to update my code in the future, they're going to have a hard time.
- They don't scale well to large or complicated inputs.
- The rules differ across various systems (again the maintainability point).

- Depending on the input, there might be better tools.

1.6. (2pts) Data persistence: Files, Databases, "other"

a. Give an example of when using server-side files (either `.txt` or `.json`) are better than using a database:

Possible answers:
- When content doesn't change.
- When content pieces are isolated from each other (don't "relate" to others)
- If you have large unstructured data.
- When you want to preserve nested structure
- Overhead of SQL outweighs benefits

b. LocalStorage and SessionStorage are nearly identical key-value database options for client-side storage. Give one example scenario for when SessionStorage is more appropriate and state why:

Possible examples:
- Storing a user login token
- Temporary information only needed while the page is open
- For performance reasons

1.7. (2 pts) File I/O in Node

a. Assume we have a Node app that has imported the filesystem module with the following line:

```
const fs = require('fs').promises;
```

One of these two snippets is right. **Circle** the one **without the error**, and then briefly explain why it's right.

```
// Example A
let txt = fs.readFile("file.txt");
txt = txt.split("/n");
res.send(txt);
```

```
// Example B
let txt = await fs.readFile("file.txt");
txt = txt.split("/n");
res.send(txt);
```

Possible answers:
- Need await because fs.readFile returns a promise.
- We're doing require('fs').promises which means everything from that module returns a promise.

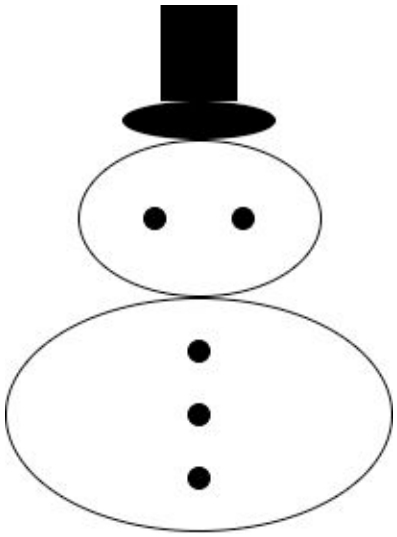
1.8 (1pt) When would an `ENOENT` error occur?

Possible answers:
- It would occur then the file or directory you are trying to read does not exist.

1.9. (5pts) Selector the Snowman

On the left is the HTML representation of the snowman on the right.

```
<body>
  <div id="snowman">
    <div id="crown"></div>
    <div id="brim"></div>
    <div id="head">
      <div class="button"></div>
      <div class="button"></div>
    </div>
    <div id="torso">
      <div class="button"></div>
      <div class="button"></div>
      <div class="button"></div>
    </div>
  </div>
</body>
```



On the lines to the right, write all of the selector(s) for which these rules would apply in order to obtain the image above:

```
border-radius: 50%;              #brim, #head, #torso, #button
display: flex;                   #snowman, #head, #torso
flex-direction: column;          #snowman, #torso
justify-content: space-evenly;   #head, #torso
align-items: center;             #snowman, #head, #torso
```

2. (28 pts) Create your own API in Node

For Problem 2, we're writing part of the backend for starting a new race and recording the winner. As this is backend Node code, you can assume for each of Part A and B that you're working within the following template:

```
"use strict";

const express = require('express');
const fs = require('fs').promises;
const util = require('util');
const multer = require('multer');
const glob = require('glob');
const globPromise = util.promisify(glob);

app.use(express.urlencoded({extended: true}));
app.use(express.json());
app.use(multer().none());

const app = express();

/*
      Part A (1) will be written here
*/

/*
      Part B (2) will be written here
*/

/**
 * Takes an array and returns an array with the same elements in a random order.
 * @param {array} array the array to shuffle
 */
function shuffle(array) { /*...*/ }

/**
 * Records a winner for the last race.
 * @param {string} driver The name of the winning driver.
 * @param {string} wagonColor The color of the winning wagon.
 * @return {boolean} true if the winner was successfully recorded, false if not
 */
function recordWinner(driver, wagonColor) { /*...*/ }

app.use(express.static("public"));
const PORT = process.env.PORT || 8000;
app.listen(PORT);
```

Problem 2, Part A(1): getWagons

```
/* Write your code for Part A(1) here */
_async____ function getWagons(numWagons) {
  let wagonFiles = await globPromise('wagons/*.txt');
  if (numWagons > wagonFiles.length) {
    return null;
  }
  let wagons = [];
  for (let i = 0; i < wagonFiles.length; i++) {
    Let wagonFile = wagonFiles[i];
    let wagonData = (await fs.readFile(wagonFile, 'utf8')).split(' ');
    wagons.push({
      "driver": wagonData[0],
      "color": wagonData[1],
      "capacity": parseInt(wagonData[2]),
      "speed": parseFloat(wagonData[3])
    });
  }
  return shuffle(wagons).slice(0, numWagons);
}
```

Problem 2, Part A(2): /new_race

```
app.get('/new_race', async (req, res) => {
  if (!req.params['num_wagons']) {
    res.status(400).json({
      "error": "Invalid parameter"
    });
  } else {
    try {
      let wagons = await getWagons(parseInt(req.params['num_wagons']));
      if (wagons === null) {
        res.status(400).json({
          "error": "Invalid parameter"
        });
      } else {
        res.json({
          "wagons": wagons,
          "distance": Math.floor(Math.random() * 5000) + 1
        });
      }
    } catch (err) {
      res.status(500).json({
        "error": "Cannot create new race"
      });
    }
  }
}
```

Problem 2, Part B: /winner

```
/* Write your code for Part B here */

app.get('/winner', async (req, res) => {
  let driver = req.body.driver;
  let color = req.body.color;
  if (!driver || !color) {
    res.status(400)
      .type('text')
      .send('Could not record winner: ' + driver + ' in a ' + color + ' wagon.');
  } else if (recordWinner(driver, color)) {
    res.type('text')
      .send('Recorded winner: ' + driver + ' in a ' + color + ' wagon.');
  } else {
    res.status(500)
      .type('text')
      .send('Could not record winner: ' + driver + ' in a ' + color + ' wagon.');
  }
}
```

3. (17pts) Racing the Client's Data

Problem 3, Part A: newRace

```
/** Initiate the fetch request */
function newRace() {
  fetch('/new_race?num_wagons=' + id('num-wagons').value)
    .then(checkStatus)
    .then(res => res.json())
    .then(startRace)
    .catch(console.error);
}

function startRace(res) {
  id('num-wagons').value = '';
  id('tracker').innerHTML = '';
  for (let i = 0; i < res.wagons.length; i++) {
    let wagon = res.wagons[i];
    placeCar(i, wagon.color, wagon.driver, wagon.speed);
  }
  activeRace = setInterval(moveCars, 500);
}
```

Problem 3, Part B: recordWinner

```
/**
 * Records the winning car!
 * @param {Element} car The car that won.
 */
function recordWinner(car) {
  let params = new FormData();
  let winner = card.id.split('-');
  params.append('driver', winner[1]);
  params.append('color', winner[0]);
  fetch('/winner', {
    method: 'POST',
    body: params
  })
    .then(checkStatus)
    .catch(console.error);
}
```

4. (15pts) Modify the DOM with some JS

Problem 4, Part A: placeCar

```
function placeCar(index, color, driver, speed) {
  let car = gen('img');
  car.classList.add('wagon');
  car.src = 'img/' + color + '_wagon.png';
  car.alt = color + ' Wagon';
  car.id = color + '-' + driver + '-' + speed;
  car.style.top = getCarTopPx(index);
  id('tracker').appendChild(car);
}
```

Problem 4, Part B: moveCars

```
function moveCars() {
  let cars = qsa('.wagon');
  for (let i = 0; i < cars.length; i++) {
    if (moveACar(cars[i])) {
      clearInterval(activeRace);
      activeRace = null;
      recordWinner(cars[i]);
    }
  }
}
```

5. (15pts) Store your data in a SQL database

Consider the following database table, inspired by the imdb database from the CSE154 Query Tester:

Table name: `movies`

id	title	rating	synopsis	studio
123	My Little Tags: The Movie	9.5	, , <aside>, <footer>, <article>, and <section> embark on an epic CSE154 journey...	netflix
456	HTML 5	5.0	<head> the HTML Head and her sister <header> embark on an adventure in...	prime
521	De Bug Log	4.2	console.log() secretly dreams during CSE154 lecture of becoming a useful debugger method...	prime
946	The Table Movie	6.5	According to all known laws of web development, the table is impossible to use as layout...	sony
468	fetch() and app.get()	9.0	Written by Ecma International early in their career, two star-crossed lovers in CSE154...	disney
154	CSE154: The Movie	8.1	Five different languages come together to accomplish something magical...	uwcse

Assume there is more data in these tables than shown.

Part A: Write your own SQL.

i.
```
SELECT title, rating FROM movies WHERE studio = 'netflix' AND rating >= 8.0 ORDER BY rating DESC;
```

ii.
```
DELETE FROM movies WHERE studio = 'disney' OR studio = 'sony';
```

iii.
```
UPDATE movies SET rating = 10.0 WHERE synopsis LIKE '%CSE154%';
```

Part B: SQL in Node.

i. Our web service APIs often require inserting or selecting from a database using user-provided values. What could a malicious user do if we don't code defensively with proper protections?

Possible answers:
- Could add unwanted SQL code.
- Could reveal secret information.
- Could modify database.

ii. Given the following code snippet, complete the query string and write the code to execute it in a way that is not susceptible to a malicious user. You may assume the correct package has been required and a `db` object has already been created appropriately.

```
const db = mysql.createPool(connInfo);

app.post("/add_movie", async (req, res) => {
  let movie = req.params["movie"]; // Assume this is not undefined.
  let rating = req.params["rating"]; // Assume this is not undefined.

  // TODO: finish this query statement
  let query = "INSERT INTO movies (title, rating) VALUES(?, ?)";

  try {
    // TODO: run the query against the database.

    await db.query(query, [movie, rating]);

    res.type("text").send("OK");
  } catch (err) {
    res.type("text").send("Error!");
  }
});
```