Web Developer Test

General Knowledge:

1) Describe a RESTful service.

   - REST is short for Representative State Transfer.
   - A RESTful API will have An identifier for the resource, also called an endpoint, as well as the operation to perform on that resource. The operations are often called CRUD: Create, Read, Update, and Delete, and are represented by the HTTP “verbs”: POST, GET, PUT, and DELETE, respectively, though you can also use PATCH for updating.
   - The operations are “stateless” meaning all of the data required to do the operation is in the request.
   - Optionally, a request to a RESTful Server can include a body with the data necessary to do those operations.

2) Describe a SPA application.

   - A Single-Page Application (SPA) is essentially a web app/website that responds to user input by rewriting the current page with new data from the server rather than opening an entirely new page with the new data rendered on it.

Vue.js

1) What are the differences between two-way data binding and one-way data binding?

   In the context of an input element:

   - In Vue, two-way binding is done with the “v-model” keyword. When the bound data is changed, the input value will also change and when the input value changes, the bound data changes.
   - In Vue, one-way binding is done with the “v-bind” keyword. When the bound data is changed, the input value will change, but if the input value changes, the bound data does NOT change.

2) What is Vuex? What problems does it try to solve?

   - Vuex is a “State Management Pattern” library for Vue. It acts as a shared data store for the various components in an app. One of the primary problems it solves is that is allows transfer of data between components that do not have a parent-child relationship (and thus the data cannot be passed directly using props).

JavaScript

1) Write a function to remove duplicate items from an array of strings.

```
const removeDupes = (strings) => {
  let unique = {};
  for (let string of strings) {
    if (unique[string] === undefined) {
      unique[string] = true;
    }
  }
  return Object.keys(unique);
}
```

2) What is the output of the following, and explain why?

```
let object1 = { elem: 1 };
let object2 = { elem: 1 };
let object3 = object1;
console.log(object1 === object2);
console.log(object1 === object3);
```

- The first console log will print false, and the second console log will print true.
- This is because === checks for reference equality when used for objects.
- Though object1 and object2 appear equal, the variables actually point to different objects, so object1 === object2 is false, conversely, object1 and object3 both point to the same object, thus object1 === object3 is true.

3) Given the following

```
function wait(label) {
  return new Promise(resolve => {
    setTimeout(resolve(label), 1000);
  });
}
async function test(){
  const list = [wait('First'), wait('Second')];
  console.log('Started function');
  list.forEach(async function(x){
    console.log('Started');
    console.log(await x);
    console.log('Finished');
  });
  console.log('Finished function');
}
```

We want the output to look like

1. Started function
2. Started
3. First
4. Finished
5. Started
6. Second
7. Finished
8. Finished Function

What is the output and how would you fix the function to operate in the assumed manner?

- Assuming you called test(), the output would be:

```
Started function

Started

Started

Finished function

First

Finished

Second

Finished
```

- If we assume that we want the whole output printed at once, one way to fix the function would be to remove the async await from the forEach, but add await keywords before the calls to wait. EX:

```
async function test(){

  const list = [await wait('First'), await wait('Second')];
  console.log('Started function');
  list.forEach(function(x){
    console.log('Started');
    console.log(x);
    console.log('Finished');
  });
  console.log('Finished function');
}

test();
```

- If we want the outputs to display one at a time 1 second apart, we could write it like this:

```
function wait(label) {

  return new Promise(resolve => {
    setTimeout(() => resolve(label), 1000);
  });
}

async function test(){
  const list = ['First', 'Second'];
  console.log('Started function');
  for (let label of list) {
    let outputLabel = await wait(label);
    console.log('Started');
    console.log(outputLabel);
    console.log('Finished');
  }
  console.log('Finished function');
}

test();
```

- Note that I added ()=> to the wait function, because otherwise, resolve(label) runs immediately, and that I swapped the forEach with a for loop, because async await does not work well with forEach.

Performance

1) What is the Big O notation of the following functions?

```
function Problem1(num = 0){
  return 0;
}
function Problem2(array = []) {
  for(let i =0;i<array.length;i++){
  }
}
function Problem3(array = []) {
  for(let i =0;i<array.length;i++){
    for(let k=0;k<array.length;k++){
    }
  }
}

function Problem4(object ={}, haskey="") {
  if(haskey in object){
    return true;
  }
  return false;
}
function Problem5(object ={}, haskey="") {
  return Object.keys(object).includes(haskey)
}
```

- Problem 1: O(1)
- Problem 2: O(n) where n is the length of the array
- Problem 3: O(n^2) where n is the length of the array
- Problem 4: O(1), object lookup (by the “in” keyword in this case) is constant
- Problem 5: O(n), assuming the includes function is a linear search (which the built in one is)

2) Which Method is more performant? Explain your reasoning.

```
function Method1(arr = [],search =""){
  return arr.find(x=>x === search) !== null
}
function Method2(obj = {},search =""){
  return Object.keys(obj).find(x=> x === search) !== null
}
function Method3(obj = {},search =""){
  return (search in obj);
}
function Method4(arr = [],search =""){
  return arr.includes(search)
}
```

- Method 3 is the most performant.
- Array.find checks every element in the array to see if the callback returns true, so it is O(n), so Methods 1 and 2 are O(n). (With method 2 taking longer because it has to convert the object keys into an array)
- Array.includes is a linear search, so it is at O(n), and thus method 4 is O(n).
- The in operator for objects is just object lookup, so it is O(1), thus Method 3 is O(1).

SQL

You have three tables

- Card (card_id, name)
- Deck (deck_id, card_id)
- Game (game_id, deck_id)

Write SQL queries to answer the following questions. You may use any SQL dialect you wish.

Using **PostgreSQL**, (note, in postgres, it is better to have all lowercase table names so quotes are not necessary)

1) Insert a Card
   - Assuming that card_id is an integer and does NOT auto-increment
   - `INSERT INTO “Card” (card_id, name) VALUES (0, ‘1 of Spades’);`
2) Update a Card Name
   - `UPDATE “Card” SET name = ‘1 of Clubs’ WHERE card_id = 0;`
3) Delete a Game
   - Assuming game_id is an integer, and that only game is deleted, not associated deck
   - `DELETE FROM “Game” WHERE game_id = 2;`
4) Find the most used Card in all Games.
   - Assuming that the Deck table lists all cards for all games. For example, the table could include both row (0, 1) and (0, 2), meaning the deck with id 0 contains cards with the ids 1 and 2.
   -
```
SELECT card_id AS most_common_card
FROM “Deck”
GROUP BY card_id
ORDER BY COUNT(*) DESC
LIMIT 1;
```

Programming

Please implement the following. This will be judged based on **Readability** and **Functionality**.

1) You have an object of account's given in the format:

```
const accountData = {
  '33fb0cb9-b24e-4f11-b16c-ed9055a78459':{
    'createdAt':1005,
    'name':'Player1'
  },
  'e1e4624e-dd90-4009-9fc6-9cad750b8fda':{
    'createdAt':79,
    'name':'Player2'
  },
  '6d0153b1-9522-4a5b-b793-e6185b5e4dbe':{
    'createdAt':'5000',
    'name':'Player3'
  },
  '3d914cd1-5163-4678-b081-053f6a7fbd64':{
    'createdAt':92,
    'name':'Player4'
  },
}
```

1. Write a function to return the name of the biggest createdAt value.

   - Assuming ‘createdAt’ is a positive integer. Returns empty string if accountData is an empty object

   -
     ```
     const biggestCreatedAt = (accountData) => {
       let biggestCAName = ‘’;
       let biggestCA = -1;
       for (let id in accountData) {
         if (accountData[id].createdAt > biggestCA) {
           biggestCAName = accountData[id].name;
           biggestCA = accountData[id].createdAt;
         }
       }
       return biggestCAName;
     }
     ```

2. Write a function to return the name of the smallest createdAt value.

   - Assuming ‘createdAt’ is a positive integer. Returns empty string if accountData is an empty object

   -
     ```
     const smallestCreatedAt = (accountData) => {
       let smallestCAName = ‘’;
     ```

```
  let smallestCA = Infinity;
  for (let id in accountData) {
    if (accountData[id].createdAt < smallestCA) {
      smallestCAName = accountData[id].name;
      smallestCA = accountData[id].createdAt;
    }
  }
  return smallestCAName;
}
```

3. Write a function to return an array of sorted accounts by createdAt (should be the smallest first)

- I added the id code as a prop of the account objects (created copy of accountData so that original would not be mutated)

-
  ```
  const sortAccounts = (accountData) => {
    let accountDataCopy = JSON.parse(JSON.stringify(accountData));
    let accountsArray = [];
    for (let id in accountDataCopy) {
      accountDataCopy[id].id = id;
      accountsArray.push(accountDataCopy[id])
    }

    accountsArray.sort((a, b) => {
      return a.createdAt - b.createdAt;
    });

    return accountsArray;
  }
  ```

2) Q. Simple Card Game.

Will be in a separate javascript (js) file called SimpleCardGame.js

1. Create a Deck
   - The deck should contain 52 unique Cards.
   - Cards are standard poker cards.
   - Suit Order
     - Hearts, Diamonds, Clubs then Spades
   - Card Values
     - Aces are 15 points.
     - Face cards (Jack, Queen, King) are 10 points.
     - Numbered cards are worth the value on the card (i.e. a 5 is 5 points).
   - The Deck should be able to be shuffled
2. Create a Player
   - The Player should have a Hand that can hold 5 Cards from the Deck.
   - The Player's Hand should be sorted by suit and then by point value
   - The Player's Hand should be able to be totaled by point value.
   - Players should be able to draw a card from the top of the deck.
   - Players should be able to print the cards in their hand to console
3. Program Flow
   - Initialize and Shuffle the Deck.
   - Create a random number of Player's between 2-6.
   - Allow each player to draw a Card from the Deck until all players have 5 cards.
   - Count the total points in each player's hand and print all players hands to console.
   - Print the Player with the most Point's (if there is a tie randomly pick one of the winning players).

Quiz video game

Will be in a separate folder called quiz_video_game.

Description

You have to make a quiz application

Requirements

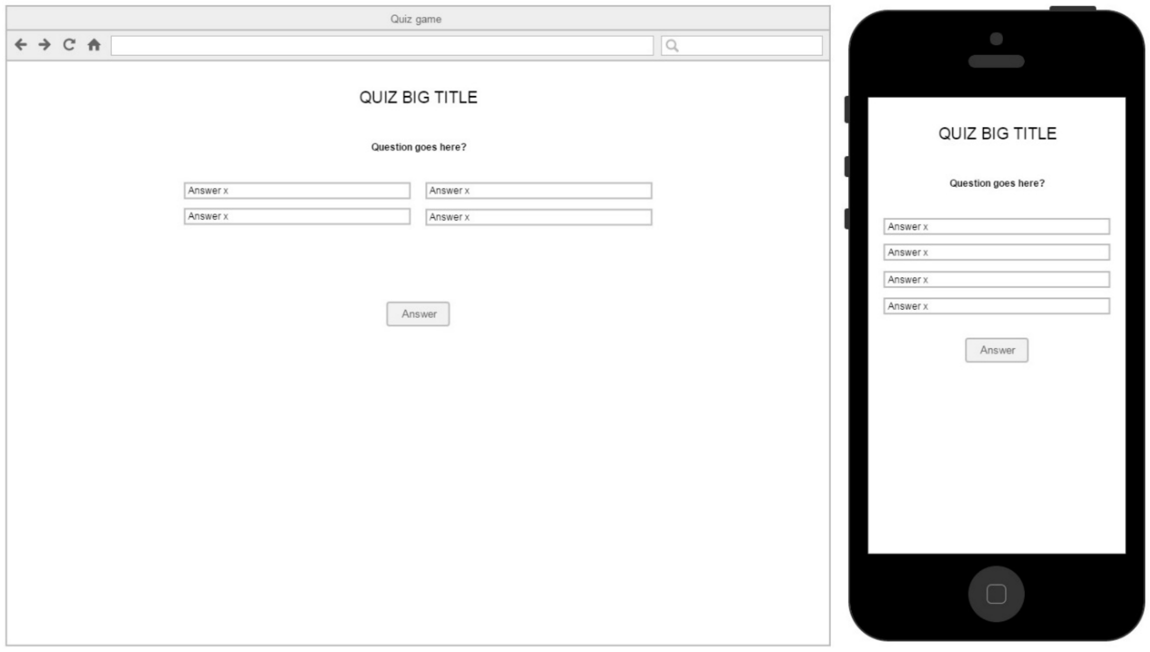
Front End Requirements

- The player can see one question at the time
- The player can see their score at the end
- The order of the questions and answer choices should be randomized
- The game should be a single-page application
- You can use Bootstrap, Materialize, or another framework as desired

Back End Requirements

- Create a nodejs backend server to serve data to the front end
- You cannot edit the quizQuestions.txt file, instead create your own data structure to use.
- The client should not know what the correct answers are.
- The server should handle question verification, the client should send a potential answer to the server for the server to verify if that answer was correct.
- The server should respond with whether the selected answer was correct or not and the client should react based on the response.

Example Quiz Layout


Quiz game
QUIZ BIG TITLE
Question goes here?
Answer x
Answer x
Answer x
Answer x
Answer
QUIZ BIG TITLE
Question goes here?
Answer x
Answer x
Answer x
Answer x
Answer