Rick Kabuto

Design RESTful web services for a rideshare service to handle the following use-case:

Your answer should include the following:

- **Identifying the resources used by the web services.**
- **The URL, method, status for each web service.**
- **A brief description of the data sent/received by each web service.**

You may assume that the rideshare service is tracking the current GPS location of both driver and passenger using services which are not covered by your answer. Your answer can ignore the many details of practical rideshare services like payment, maps, authentication and authorization which were not mentioned in the above use-case. ***20-points***

a. A passenger requests a ride to be scheduled ASAP from some location to a destination
   a. The primary resource used by the web service would be book_ride, which represents the creation of a new ride resource in the system.
   b. The service would use the POST method at the specified URL to indicate that a new ride has been requested, returning an HTTP status code of 201 to confirm that the ride resource (book_ride) was successfully created.
   c. The data sent to the service would include the rider's identity, pickup location, and drop off destination. Upon processing the request, the system would set the ride's initial status to waiting or pending, and return information about the assigned driver along with the current status of the ride.

b. The rideshare service displays a price and estimated pickup time if there are available drivers.
   a. We would use the book_ride resource to check for available drivers and, if any are found, calculate the price and estimated pickup time.
   b. The web service would use the GET method at the endpoint /book_ride/{ride_id}/estimate. This request would retrieve the estimated price and pickup time for the specified ride. The service would return an HTTP status code of 200 OK if successful, or 404 Not Found if the specified ride does not exist.
   c. The data sent and received would include the estimated ride price, the estimated pickup time, and an indication of whether a driver is available.

c. **The passenger decides to go ahead with requesting the ride.**
   a. At this point, the ride's status would be updated from pending to confirmed using the book_ride resource.
   b. The web service would use either the POST or PATCH method at the endpoint /book_ride/{ride_id}/confirm. POST would be used if the confirmation is treated as a separate action on the ride. PATCH would be used if we are simply updating the status of the existing ride resource.In either case, the service would return an HTTP status code of 200 OK if the ride was successfully confirmed, or 404 Not Found if the specified ride ID does not exist.
   c. The data sent and received would include the updated ride status and confirmation that the ride is now progressing.

d. The driver picks up the passenger
   a. This step would involve both the book_ride and driver resources, where a driver is selected and associated with the passenger who requested the ride.

b. The web service would use the PATCH method at the endpoint /book_ride/{ride_id}/assign-driver. It would return an HTTP status code of 200 OK upon successful assignment, or 404 Not Found if the driver is unavailable or the ride ID does not exist.
c. The data sent and received would include the driver's ID and the updated status of the ride indicating that the driver has been assigned and is en route.

e. The driver arrives at the destination and drops off the passenger.

a. The book_ride resource is used to track whether the driver has arrived at the destination and whether the passenger is in the vehicle.
b. The web service would use the PATCH method at the endpoint /book_ride/{ride_id}/pickup.It would return an HTTP status code of 200 OK if the update is successful, or 404 Not Found if the ride ID is invalid.
c. The data sent and received would include a timestamp and the updated status indicating the current progress of the ride.

f. The driver arrives at the destination and drops off the passenger.

a. We would use the book_ride resource to indicate the completion of the ride and update its status accordingly.
b. The web service would use the PATCH method at the endpoint /book_ride/{ride_id}/complete.It would return an HTTP status code of 200 OK if the update is successful, or 404 Not Found if the ride ID is invalid.
c. The data sent and received would include the timestamp of when the passenger was dropped off, the updated completed status, and a summary of the route taken.

2. How would you set up an individual property of an object to be invisible when printing the object.

```
> const obj = ... ;
undefined
> obj.a
42
> obj.a = 33
33
> obj
{}
> obj.a
33
```

Specifically, fill in the ... above. *10-points*

Answer) To make an individual property of an object invisible when printing the object, we can use the built in Object.defineProperty function. By setting the enumerable attribute of the property to false, we ensure that the property does not appear during operations like console.log or JSON.stringify, effectively making it invisible when printing the object.

Implementation

```
const obj = {};
Object.defineProperty(obj, 'a', {
  value: 42,
  writable: true,
  enumerable: false,
  configurable: true
});
```

---

3. Debugging JavaScript code using **`console.log()`** statements is painful when given a sequence of chained calls:

```
const result = someArray
  .filter(e => e%2 === 0)
  .map(e => e + 2)
  .filter(e => e < 10)
  .reduce((acc, e) => acc + e, 0);
```

If you would like to `console.log()` the output of the first **`filter()`** call you will have to change the code to something like:

```
const arr1 = someArray
  .filter(e => e%2 === 0);
console.log(arr1);
const result = arr1
  .map(e => e + 2)
  .filter(e => e < 10)
  .reduce((acc, e) => acc + e, 0);
```

What would be useful is to have something like:

```
const result = someArray
  .filter(e => e%2 === 0)
  .tap(e => console.log(e))
  .map(e => e + 2)
  .filter(e => e < 10)
  .reduce((acc, e) => acc + e, 0);
```

where **`tap(f)`** can be chained off any **`Object`** and returns **`this`**. If it is called with an argument which is a single argument function, then that function is called with `this` as its actual argument.

Example:

```
> Array.from({length: 10}).tap(x => console.log(`original ${x}`))
   .map((_, i) => i+1).tap(x => console.log(`mapped ${x}`))
   .filter(e => e > 5).tap(x => console.log(`filtered ${x}`))
   .reduce((acc, v) => acc*v, 1)
  original ,,,,,,,,,
  mapped 1,2,3,4,5,6,7,8,9,10
  filtered 6,7,8,9,10
  30240
>
```

Answer) I am quite unsure what this question is asking? There is really nowhere in the directions of this question that explicitly says what I need to do. I am assuming you want me to implement a function/method that allows us to use the .tap strategy in the example. This is my implementation

```
Object.prototype.tap = function(call) {
  if (typeof call === 'function') {
    call(this);
  }
  return this;
};

const result = Array.from({ length: 10 })
  .tap(x => console.log(`original ${x}`))
  .map((_, i) => i + 1)
  .tap(x => console.log(`mapped ${x}`))
  .filter(e => e > 5)
  .tap(x => console.log(`filtered ${x}`))
  .reduce((acc, v) => acc * v, 1);

console.log(result);
```

Provide HTML for each of the following form controls:
Your answers should:

- **Be set up to maximally constrain the user's input to legal values and minimize the opportunity for error.**
- **Specify at most a single HTML control for each of the above and should provide the actual HTML code for each control (if a control allows only certain discrete values not specified above, then your HTML code can contain some typical values).**
- **If there are multiple candidates for a specific control, discuss the trade-offs between the candidates.**

10-points

A) A control which allows the user to provide a person's age.

- <input type = “number” name=”age” min=”0” max=”120” required>

B) A control which allows the user to select one of several show times for a movie.

- <select name="movieTime" required>
  <option value="">Choose a time</option>
  <option value="6pm">6:00 PM</option>
  <option value="9pm">9:00 PM</option>
  </select>

C) A control which allows the user to enter in a credit card number.

- <input type="text" name="cardNumber" inputmode="numeric" pattern="\d{13,19}" required>

D) A control which allows the user to control the speed of an animation.

- <input type="range" name="speedControl" min="0.5" max="3" step="0.1" value="1">

E) A control which allows the user to enter a social security number.

- <input type="text" name="ssnInput" pattern="\d{3}-\d{2}-\d{4}" placeholder="123-45-6789" required>

5. Given a page containing tables listing information about different computer models with an example table like:

```
<table id="computer-models">
  <tr>
    <th class="time">Model</th>
    <th class="mercury">Clock Speed</th>
    <th class="price">Price (USD)</th>
  </tr>
  <tr class="ln-2600">
    <td class="model">LN 2600</td>
    <td class="clock-speed">2.6 GHz</td>
    <td class="price">849.99</td>
  </tr>
  <tr class="ln-2100">
    <td class="model">LN 2100</td>
    <td class="clock-speed">2.1 GHz</td>
    <td class="price">749.99</td>
  </tr>
  <tr class="ln-3200">
    <td class="model">LN 3200</td>
    <td class="clock-speed">3.2 GHz</td>
    <td class="price">999.99</td>
  </tr>
</table>
```

Subject to the same restrictions (no destructive operations or recursion) as the programming exercises in Homework 1, provide code for the following functions:

Subject to the same restrictions (no destructive operations or recursion) as the programming exercises in Homework 1, provide code for the following functions:

A) A function modelRowElement(tableId, modelId) which returns the DOM element corresponding to the <tr> element for the model specified by modelId in the table specified by tableId.

```
function modelRowElement(tableId, modelId){
  const x = document.getElementById(tableId);
  if(!x)
    return null;
  return Array.from(x.getElementsByTagName('tr')).find(row => row.classList.contains(modelId)) || null;
}
```

B) A function modelSpeedElement(tableId, modelId) which returns the DOM element corresponding to the <td> element for the clock-speed for the model specified by modelId in the table specified by tableId.

```
function modelSpeedElement(tableId, modelId) {
  const x = document.getElementById(tableId);
  if (!x)
    return null;
```

```
    const y = Array.from(x.getElementsByTagName("tr"))
                .find(r => r.classList.contains(modelId));
    if (!y)
        return null;
    return Array.from(y.getElementsByTagName("td")).find(td => td.classList.contains("clock-speed")) || null;
}
```

C) A function modelSpeed(tableId, modelId) which returns the JavaScript Number giving the value for the clock-speed for the model specified by modelId in the table specified by tableId.

```
function modelSpeed(tableId, modelId) {
    const speedCell = modelSpeedElement(tableId, modelId);
    if (!speedCell)
            return null;
    const match = speedCell.textContent.match(/\d+(\.\d+)?/);
    if (!match) return null;
             return Number(match[0]);
}
```

D) A function clockSpeeds(tableId) which returns a JavaScript list containing all the clock-speed Number values in the table specified by tableId.

```
function clockSpeeds(tableId) {
    const x = document.getElementById(tableId);
    if (!x)
        return [];
    const y = Array.from(x.getElementsByTagName("tr")).slice(1);
    return y.map(row => {
        const cell = row.querySelector('.clock-speed');
        if (!cell)
            return null;

        const text = cell.textContent.trim();
        const val = parseFloat(text);
        return isNaN(val) ? null : val;
    }).filter(val => val !== null);
}
```

E) A function averagePrices(tableId) which returns the average price of all the computer models in the table specified by tableId. It should return 0 if there are no prices.

```
function averagePrices(tableId) {
    const x = document.getElementById(tableId);
    if (!x)
        return 0;
    const y = Array.from(x.getElementsByTagName("tr")).slice(1);
    const z = y.map(r => r.querySelector(".price"))
              .filter(p => p !== null)
              .map(p => {
                 const m = p.textContent.match(/\d+(\.\d+)?/);
                 return m ? Number(m[0]) : null;
              })
              .filter(v => v !== null);
```

```
    if (z.length === 0)
        return 0;
    const total = z.reduce((a, b) => a + b, 0);
    return total / z.length;
}
```

With modern web applications, it is preferable to validate all user input as soon as possible once a user provides the input. Discuss how you would validate each of the following user inputs:
Your answer should provide:

- **The details of the validation performed.**
- **The DOM events you would use to trigger validation.**
- **A discussion of what validations can be performed within a browser and which validations should be performed on the server.**

A US zip code.

- The validation should be performed using a regular expression pattern to check whether the input is in the correct 5-digit ZIP code format or in the "ZIP+4" format.
- The DOM events I would use to trigger validation are input, blur, and submit. I would use real time input highlighting to indicate errors as the user types, trigger
- Validations should occur in the
  - Server
    - To confirm that the input received meets the correct format and hasn't been bypassed.
  - In Browser
    - To ensure the format is correct using a regular expression. Submission should be blocked if the format is invalid, and the user should be notified immediately.

A US address.

- The validation should be performed making sure that there is a street number and name. We can do this using a regular expression again
- The DOM events we would use to trigger the validation are input, blur, and submit. The input event helps catch errors as the user types, blur checks the field when the user leaves it, and performs a final validation before the form is sent to the server.
- Validations should occur in
  - Server
    - The server can make sure the address format is correct and whether the entered address corresponds to a real, existing location.
  - In Browser
    - The browser makes sure that the address field is not left empty and that the input starts with a number followed by a street name.

A credit card number.

- We can use Luhn's algorithm, a simple checksum formula used to verify that the credit card number has been entered correctly.
- The DOM events used to trigger validation are input, blur, and submit. The input event helps detect formatting issues, the blur event performs a final validation when the user exits the field, and the submit event ensures the credit card number passes all validation checks.
- Validations should occur in
  - Server
    - Server side validation should check the Luhn algorithm to confirm the number is valid, check for signs of fraud, and securely handle the credit card information such as encrypting it before storing or transmitting.

- In Browser
  - Validation in the browser should check that the credit card number is in the correct format and that it passes the Luhn algorithm.

7. The tests provided for Project 3 use supertest to test the web services.

   Look sufficiently into the workings of **supertest** to explain how the test shown below will work.

```
const BASE = '/api';

describe('grades web services', () => {

  ...
  let ws: ReturnType<typeof supertest>;

  beforeEach(async function () {
    ...
    dbGrades = servicesResult.val;
    const app: App = serve(dbGrades, { base: BASE}).app;
    ws = supertest(app);
  });

  ...
  describe('add global student', () => {

    it('must successfully add a student', async () => {
      await addStudent(ws, StudentsData[0]);
    });
    ...
  });
  ...
});

...
async function addStudent(ws: ReturnType<typeof supertest>,
                          student: T.Student) {
  const url = `${BASE}/students`;
  const res =
    await ws.put(url)
     .set('Content-Type', 'application/json')
     .send(student);
  expect(res.status).to.equal(STATUS.CREATED);
  expect(res.body?.isOk).to.equal(true);
  ...
  return res;
}
```

   Specifically show how **supertest** sends the request to the wrapped express server and is able to test the response. You may want to look at the source code for **supertest** and its dependencies. Tracing the test could also be useful. *20-points*

The first that occurs is that supertest will wrap and Express app in a test agent and this will allow the HTTP requests to work directly towards the stack and bypassing the network communication using ws = supertest(app); Next, we are going to send a request and create a mock HTTP PUT request to /api/students and we use a superagent to build and manage this request.

```
const url = `${BASE}/students`;
const res =
  await ws.put(url)
   .set('Content-Type', 'application/json')
   .send(student);
```

Next, supertest is going to use the .handle to invoke the Express app with a fake request and response object which allows us to process the request as if it came from a real HTTP client from const app: App = serve(dbGrades, { base: BASE}).app; Finally, superest will test the response and check the test with the objects. These will create Assertions which verify that the server responded correctly with the HTTP status and JSON structure.

```
expect(res.status).to.equal(STATUS.CREATED);
expect(res.body?.isOk).to.equal(true);
```