HomeWork-3

Answer 1:

The Typescript playground handbook clearly indicates the structure of the url. The hash generally represents the state of the editor:

```
The hash generally represents the state of the editor:

- `#code/PRAz3dDc3...` - A base64 and zipped version of the code which should live in the editor.
```

In our example link:
https://www.typescriptlang.org/play?#code/PTAqGUAIEMBsH...

The part after #code/ represents the base64 and zipped version of the code in editor.

To implement links to programs typed in by its users typescript follows the below approach:

- The code in the editor is first serialized into a string i.e. the code text present in the editor is converted into a long string.
- Then it uses the lzstring to compress the code and append to the url. This is done using the compressToEncodedURIComponent method as shown below:

```
787
788     const zippedCode = lzstring.compressToEncodedURIComponent(originalCode)
789     const playgroundURL = `https://www.typescriptlang.org/play/#code/${zippedCode}`
790
```

- Similarly decompressToEncodedURIComponent is used when accessing the link.

For the compression above Lempel-Ziv compression algorithm is applied to this string first which takes care of the repeated occurrences of data in the string.

After compression the result is checked for safety by replacing special characters with safe alternatives and then encoded to base 64. Reverse process is followed while decompression.

Answer 2:

Requirement: Setting up a property in the object so as to be able to export the object to clients while making it impossible for clients to access that property within JavaScript.

This can be done using function as constructor and closure: Normally a constructor function does not explicitly return a value. In that case, the return value is set to a reference to the newly created object. Any function which is invoked preceded by the new prefix operator is being used as a constructor as shown in the below example:

```
> function ConstructorFunction1(x, y) {
      let x1 = x;
      this.y1 = y;
  }

  let obj = new ConstructorFunction1('test1', 'test2');
```

In the above implementation, the function is used as a constructor to create a object using new keyword. If the object is exported, the value of y1 can be accessed but the instance variable x1 is private and cannot be accessed using the object.

We can use closure to allow controlled access of variable x1 by defining a public getter function as shown below:

```
> function ConstructorFunction1(x, y) {
      let x1 = x;
      this.y1 = y;

      this.getX1 = function() {
          return x;
      }
  }

  let obj = new ConstructorFunction1('test1', 'test2');
  console.log(obj.getX1());
  test1
```

Answer 3:

HTML is primarily designed for creating web pages and web applications. It is a markup language used for web content. While it is possible to use HTML as a representational format for a REST API, it is not used for this purpose. The main reason is that not all JSON datatypes can be naturally mapped also HTML is a document format, not a data format. It is designed to represent text documents with embedded links and not structured data.

- Datatype mapping (JSON -> HTML):

  String (Natural mapping) : When converting a JSON string to HTML, we can use the <p> tag for paragraphs or the tag for inline text elements.

  Number (Natural mapping) : A JSON number can also be easily converted to HTML using the <p> or tags.

  Object(Forced mapping): Converting a JSON object to HTML can be done by placing it in an HTML <table>, with each property as a row. However, this mapping is not very natural as HTML tables are not ideal for representing key-value pairs.

  Array(Natural mapping): A JSON array can be converted to an HTML <ul> or <ol> list, with each item as a list item (<li>). But if the array contains objects, the conversion becomes more complex.

  Boolean(Natural mapping): A JSON boolean can be simply displayed as a string in HTML.

  Null(Natural mapping): JSON null can be represented as an empty HTML element or a special string like ‘null’.

- Support for HATEOAS:

  ->HATEOAS is a concept in REST that enables a client to communicate with a server using hypermedia that is provided dynamically by the server.

  ->While HTML is a hypermedia format that supports links and forms, which are essential for implementing HATEOAS, it is more suitable for human consumption rather than machine-readable APIs.

-> JSON, on the other hand, is widely preferred for REST APIs due to its simplicity, compatibility with JavaScript, and ease of parsing, generation, and processing compared to HTML.

-> For representing a REST API, HTML is a good choice due to its support for HATEOAS. However, JSON is often preferred for machine-to-machine communication because of its simplicity and flexibility.

->JSON lacks the hypermedia features of HTML. So, to incorporate HATEOAS into JSON, extra conventions and standards may be needed.

Answer 4:

Below are the specifications for each of the required web-services:

1. Searching for restaurants in the chain:

- Suitable HTTP method: GET
- URL: /findRestaurants
- Input Parameters: Query parameters such as name, location
- Response: A list of all restaurants that match the search criteria. Each restaurant could include a link to its own details (e.g., /findRestaurants/{id}).
- Caching: Responses can be cached based on search parameters. The cache can be public since anyone can search for the restaurants and the restaurant details are not likely to change soon so age can be long depending on attributes of restaurant object. Example: Cache-Control: private, max-age=72000, must-revalidate

2. Making a reservation at a particular restaurant:

- Suitable HTTP method: POST
- URL: /findRestaurants/{id}/reservations
- Input Parameters: Request body containing details of reservations, example: {customer_name, phone, date, time, tableSize}
- Response: A success response with reservation_id and link to the reservation details (e.g., /reservations/{id}).
- Caching: As this is a write operation, caching is not applicable.

3. Searching for reservations:

- Suitable HTTP method: GET
- URL: /reservations
- Input Parameters: Query parameters such as reservation_id, date etc.
- Response: A list of reservations that match the search criteria. Each reservation could include a link to its own details (e.g., /reservations/{id}).
- Caching: Responses can be cached based on the search criteria but reservations can change frequently so caching may not be required.

4. Modifying an existing reservation:

- Suitable HTTP method: PUT
- URL: /reservations/{id}
- Input Parameters: Request body having reservation details similar to making reservations.
- Response: A success response with reservation_id and link to the reservation details (e.g., /reservations/{id}).
- Caching: As this is a write operation, caching is not applicable.

5. Canceling a reservation:

- Suitable HTTP method: DELETE
- URL: /reservations/{id}
- Input Parameters: query parameter reservation_id.
- Response: Success message
- Caching: As this is a write operation, caching is not applicable.

Answer 5:

Below are the HTML controls that would be suitable for each of the form controls:

- Percentage of children within a school district who are classified as special needs children: A field type of number can be used with a range of min =0 and max =100 to represent the percentage constraint as shown below:

  ```
  <input type="number" id="percentage-special-need" name="
  percentage-special-need " min="0" max="100" step="0.01">
  ```

- Date and time for a pickup for a ride: We can use the datetime-local input type as shown below:

  ```
  <input type="datetime-local" id="pickup" name="pickup ">
  ```

- Name of a New York City Borough: We can use a simple dropdown list without search since there are only 5 elements as shown:

  ```
  <select id="borough" name="borough">
    <option value="">Select borough</option>
    <option value="manhattan">Manhattan</option>
    <option value="brooklyn">Brooklyn</option>
    <option value="queens">Queens</option>
    <option value="bronx">The Bronx</option>
    <option value="staten-island">Staten Island</option>
  </select>
  ```

- Select an arbitrary color: color input type can be used which provides a color picker that allows the user to select any color:

  ```
  <input type="color" id="color" name="color">
  ```

- Select several predefined colors to be mixed together: The input type here would depend on the number of colors the user is required/allowed to pick and on the variety of colors provided to the user.

  For example if there are only a few colors from which user can pick then we can use multiple checkboxes as shown:

  ```
  <label><input type="checkbox" name="colors" value="red"> Red</label>
  <label><input type="checkbox" name="colors" value="blue">
  Blue</label>
  <label><input type="checkbox" name="colors" value="green">
  Green</label>
  ```

  If the user should be allowed to pick any arbitrary colors but maximum number of colors that can be picked are few than multiple color input type can be used:

  ```
  <label for="color1">Color 1:</label>
  <input type="color" id="color1" name="color1">
  <label for="color2">Color 2:</label>
  <input type="color" id="color2" name="color2">
  <label for="color3">Color 3:</label>
  <input type="color" id="color3" name="color3">
  ```

  If user should be allowed to pick a list of colors from some predefined values, a multi-select list can be used as shown :

  ```
  <select id="colors" name="colors" multiple>
    <option value="#ff0000">Red</option>
    <option value="#00ff00">Green</option>
    <option value="#0000ff">Blue</option>
  </select>
  ```

Answer 6:

Below are the required functions:

- The <tr> DOM element for the afternoon row:

```
> function getAfternoonElement(id) {
      let el = document.getElementById(id);
      let afternoonElement = el.querySelector('.afternoon');
      return afternoonElement;
  }
```

- The <td> DOM element for the afternoon mercury pressure:

```
> function getAfternoonMercuryPressureElement(id) {
      let el = document.getElementById(id);
      let result = el.querySelector('.afternoon .mercury');
      return result;
  }
```

- The JavaScript Number giving the value for the afternoon mercury pressure:

```
> function getAfternoonMercuryPressure(id) {
      let el = document.getElementById(id);
      let result = el.querySelector('.afternoon .mercury');
      let pressure = parseFloat(result.innerText);
      return pressure;
  }
```

- A JavaScript list containing all the mercury pressure Number values:

```
> function getAllMercuryPressures(id) {
      let el = document.getElementById(id);
      let mercuryPressures = el.querySelectorAll('.mercury');
      let values = Array.from(mercuryPressures).map(i =>
  parseFloat(i.innerText));
      return values;
  }
```

- A JavaScript array containing all the <td> DOM elements for the afternoon row:

```
> function getAfternoonElements(id) {
      let el = document.getElementById(id);
      let afternoon = el.querySelector('.afternoon');
      let afternoonElements =
  Array.from(afternoon.querySelectorAll('td'));
      return afternoonElements;
  }
```

Answer 7:

We can perform the required validations as explained below:

Sure, let's discuss how we can validate each of the user inputs you mentioned:

1. US Zipcode:

- A valid US zipcode is either a 5-digit number or a 9-digit number in the format `12345-6789`. We can use a regular expression to validate this format: /^\d{5}(-\d{4})?$/.
- This validation can be performed on server-side. The DOM events 'input' or change can be used to trigger the validation.
- There could be additional server side validation required as per the business requirements. This can be done to validate the area to which the zipcode belongs and will require an external api support.

2. US Address:

- Validating a US address requires both client-side and server side validations.
- Server side: This involves checking that the address isn't empty and contains certain key components (street, city, state, zipcode) and has the required data format. DOM events `input` or `change` event can be used to trigger this validations.
- Server-side validations: At the server side we can implement a address scrub by using the API provided by services like USPS to validate the address.

3. Credit Card Number:

- Credit-card number validation will require both client side validations to handle user errors and server side validations to avoid potential fraud.
- Client side validations: This will basically validate the format of the input and can be triggered using DOM events input and change. Credit card numbers follow specific formats set by the card issuers. The first six digits of a card number (including the initial MII digit) are known as the Issuer

Identification Number (IIN), which identifies the card issuer. The remaining digits are unique to the cardholder.

- The Luhn algorithm, is used to validate the number. It works by doubling every second digit from the rightmost digit, subtracting 9 from any result greater than 9, and then summing all the digits. If the total modulo 10 equals 0, then the number is valid.
- Serverside validations: It involves validating the card type, matching card number with user details etc.
- The card type (Visa, MasterCard, etc.) can be determined by the length and the starting digit of the card number. For example, Visa cards typically start with a 4 and have 13 or 16 digits, while MasterCard cards start with 51 through 55 and are always 16 digits long.
- Validating if the user details and the card holder details might also be important. This can be done by utilizing external APIs provided by banks or payment gateways.