UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIAS DE COMPUTAÇÃO

SCC0219 - Introduction to Web Development

Exercises 8: Node.js

Upload a zip file named `yourName.USPnumber.zip`, and include the files for the questions.

1. Create a simple web server that saves values (objects) indexed by string keys using Node.js. These string keys do not have blanks or special characters in them. The server will have just 3 operations: read, write, and delete:
    a. Read: returns the value associated with a key. If the key is not valid, it returns "".
    b. Write: receives a key and an object (in JSON) and writes it in the database.
    c. Delete: receives a key and deletes the key-value pair that it refers to.

    These 3 operations should use the same route: `/store`. You should separate them using the appropriate HTTP method for each operation (GET, PUT, or DELETE). Any necessary parameter must be included using the appropriate technique (ex: query strings). For instance, a GET request for the value of key `john` would be:

    ```
    http://localhost:3000/store/john
    ```

    The key-value pairs can be recorded in a variable on the server using a Javascript object (remember that all Javascript objects are maps). For instance:

    - PUT request:
        URL: `http://localhost:3000/store/john`
        Body: `'{ "name": "John", "surname": "Doe"}'`
        Will save the record in the JavaScript variable `db`:
        `db['john'] = '{ "name": "John", "surname": "Doe"}'`
    - GET request:
        URL: `http://localhost:3000/store/john`
        Will return:
        Body: `'{ "name": "John", "surname": "Doe"}'`

    There is no need to use a database or save the object to disk. There is no authentication. Any client can read, write, or delete.

    Tips:
    1. Follow the video course up to Lesson 10 and you will end up with a running web server. In lessons 11 and 12, the code is refactored to be more organized.
    2. Lesson 10 explains the REST actions (GET, PUT, POST, and DELETE) and shows how to test the requests using the Postman tool (install in lesson 3).
    3. After lesson 10, you have a running server that has GET, PUT, and DELETE services.
    4. Which services do you need for your system? The server already has operations for these 3 services. For the PUT, it receives the id, but for the GET, it does not. Use the PUT route as a model and modify the GET route to receive an id too. Test.

5. Right now, the server just sends back the status information. Modify the router implementations to store and send back the information you need. Remember that you can only receive and send back text to the client. Use `JSON.parse()` and `JSON.stringify()` functions to convert JavaScript objects to and from text whenever needed.

(6 pts)

2. The `test.html` file (Annex 1) has a vue.js application that reads information about users, using the JSON format, from a mocked server (a server that simulates web services). To see this file, open it in a browser. It works but only implements the read (GET) functionality. Modify the file to implement the `read()`, `create()`, and `delete()` functions using the web server that you wrote for the last question. Only these 3 functions need to be modified/created. Use the username as the key to save the information in the web server. To serve the `test.html` file, create a directory called `public` in your web server and use the command

```
app.use(express.static('public'))
```

to serve the `test.html` file as `http://localhost:3000/test.html`. There is no need to modify anything else in the web server.

(4 pts)

Challenger (2pts)

3. Create a user registration system using a Node.js backend and a Vue.js client. This system must have:
   a. A login vue.js page with:
      - A login panel with username and password fields and a login button. If the user supplies a correct username/password, it opens the main page. If not, the application will popup an error message.
      - A panel to create new users. In it, users must provide a username, password, email, and address to create new system users. Optionally, that panel can be implemented as another page, and, in this case, the login page will have a button to jump to it. After creating a new user, the system returns to the login page.
   b. The main vue.js page with:
      - The user information: username, password, email, and address.
      - A button to delete the user. When a user is deleted, a popup message is shown and the system returns to the login page.
      - A button to return to the login page.

   Both vue.js pages can be implemented using basic components or, if you want, you can use the login interface you are writing for your Group Assignment. The code created here can be reused in the Group Assignment project. For security reasons, the server (not the client) should check:
   - If the username has no spaces or upper case characters and if it is greater than 5 characters and smaller than 9.
   - If the email is properly formatted (`<something> @ <something>`).

- If the password is, at least, bigger than 8 characters and has no spaces.
- If the address has, at least, 10 characters.

In case of errors, an error message will popup (stating the kind of error) and no new user will be created. There is no need to delete the problematic field(s), the user can do that.

Before coding, think about which services the server has to have. Give them URLs and decide which arguments they will need and which information they will send back.

Notice:

- There is no need to create an actual user section, using cookies or other technologies.
- Data can be recorded in variables on the server. There is no need to use a database or save these variables to disk.
- To communicate data between client and server, you may use any data format, but an easy option is to use JSON (look for `JSON.parse()` and `JSON.stringify()` functions).

Tips:

- For the client, you may reuse the forms app from past exercises.
    a. Modify it to read the information you need.
    b. You will need two pages. You may use the Vue.js Router (Vue video lesson 32 or/and code example), but can also use div tags to change what content is being shown on the page (Ex: `<div style="visibility: hidden">`.
    c. To communicate with the server, use `this.$http.get` (.get, .post, .put) as shown at Vue video lesson 18.
- Implement the format checks (for username, email, etc) and the error messages in the server and client.

(+2 pts)

Anexo 1 (`test.html`):

```
<!DOCTYPE html>
<html>
  <meta>
    <meta charset="UTF-8">
    <title>User Info</title>
  </meta>

  <body>

    <div id="app">
      <h2>User Information</h2>

      <label>Username</label> <input v-model="username" type="text">
      <br> <br>
      <label>Password</label> <input v-model="password" type="text">
      <br> <br>
      <label>Email</label>    <input v-model="email"    type="text">
      <br> <br>
      <label>Address</label>  <input v-model="address"  type="text">
      <br> <br>
      <input type="button" value="Read" @click="read">
      <input type="button" value="Delete" @click="read">
      <input type="button" value="Create" @click="read">
    </div>

    <script
src="https://cdnjs.cloudflare.com/ajax/libs/vue/2.1.6/vue.min.js">
    </script>

    <script>
      new Vue({
        el: "#app",
        data: {
            username: "john",
            password: "",
            email: "",
            address: ""
        },
        methods: {
```

```
            read: async function() {
               try {
                 let resp = await
fetch("http://www.mocky.io/v2/5ecc1c79300000fceadddb15");
                 resp = await resp.json();
                 console.log("resp: "+JSON.stringify(resp));

                 this.username= resp.username;
                 this.password= resp.password;
                 this.email = resp.email;
                 this.address= resp.address;
               }
               catch (e) {alert("Error: " + e);}
            }
         }
       });
    </script>
  </body>
</html>
```

Good Work!