Assignment 6

Recommended readings:

- Lecture slides as starting literature
- MDN Links within slides for details
- https://nodejs.org/en/
- http://toolsqa.com/postman-tutorial/
- https://www.npmjs.com/package/mysql/

Note: All exercises must be solved using node.js, JavaScript/TypeScript and CSS not using additional libraries or frameworks (except the ones proposed).

Exercise 1 – Node.js Gallery Setup

Answer following questions:

a) What are RESTful Services or RESTful APIs?
b) What is the difference between HTTP methods GET, POST, PUT, PATCH and DELETE? How should they be used?

Setup the enclosed Node.js gallery API:

- Install Node.js if you haven't done so for previous exercises.
- Install Postman, an easy to use Desktop application for conveniently issuing HTTP requests and testing the gallery API (Hint: You don't need to register to be able to use it).
- Place `nodejs_gallery_server` into a convenient location on your computer.
- Open a new console window and navigate to `nodejs_gallery_server`.
- Run '`npm install`' for installing all required Node.js modules.

Open `nodejs_gallery_server` in a text editor/IDE of your choice and take a look at `server.js`:

a) Explain and reason about the structure of this Node.js server application, as well considering the files in the routes folder.
b) Which modules are used and what are their purposes?
c) What is Cross-Origin Resource Sharing (CORS)?

Setup encloses MySQL database:

- Install MySQL if you haven't done so already (Hint: XAMPP provides the option of installing the MySQL fork MariaDB).

- Start MySQL and import `db_import/galleryDB.sql` via phpMyAdmin or the console for creating and populating a new database ‘`webtech18gallery`’ .
- Complete the module ‘`db.js`’: finish ‘`initDb`’ by connecting to the database resolving/rejecting the Promise on success/failure. The database (‘`_db`’) should be accessible by any server component, simply by requiring and calling the ‘`getDb`’ function (e.g. see ‘`gallery.js`’).
- Test your implementation by opening a console window, navigating to `nodejs_gallery` and running ‘`npm start`’. Expected output:

  ```
  Database is connected ...
  Listening on port 3000...
  ```

Exercise 2 – Simple Authentication

Leave the server running, open up Postman and issue a simple GET request to ‘`localhost:3000`’. What is the output and why? What changes would have to be made in order to pass and output a simple parameter to the API (e.g. calling ‘`localhost:3000/hello`’)? Change the server’s default route (“/”) response to also returning such a passed parameter, e.g. “`Welcome to gallery server 1.0 – you passed the parameter hello.`” How can you additionally set a response status using express? Hint: Always kill (Ctrl+C) and restart the server using ‘`npm start`’ for changes to take effect when altering the server code.

Complete the module ‘`login.js`’ by finalizing the contained POST route, which should query the previously imported database, checking whether the password provided as JSON body *`{pass: "mypwd"}`* is correct for the user identified by the parameter ‘`:email`’. Valid email/password combinations can be determined by inspecting the database of ‘`galleryDB.sql`’ (Hint: For the sake of simplicity, the database stores passwords in plain text – never do this for serious projects, use salted hashing!). Upon success/failure return appropriate messages in JSON format to the client (see below) and set following response status codes depending on the MySQL query outcome:

- `400`: an error occurred
- `401`: login failed
- `200`: login successful

Additionally, create a rudimentary session token for a successfully logged in user by generating a random integer number and updating the user’s token column accordingly. Finally, a successfully logged in client should receive first-, lastname and token as a response (JSON Body). Test and debug your implemented POST route with Postman (e.g. ‘`localhost:3000/login/sandy@nomail.com`’). JSON Body: *`{pass: "12345"}`*

Secure the other yet unimplemented routes by completing ‘`check_auth.js`’. This module exports a function that is called before proceeding with any protected routes (see modules ‘`gallery.js`’ and ‘`image.js`’). It should receive previously created token passed via the ‘`Authorization`’ header (`req.headers.authorization`) and verify if it is still valid, i.e. the same number, before allowing the request to proceed, which is simply accomplished by calling ‘`next()`’. Hint: Since ‘`req`’ is passed on to the following route, you can use this object in order to store user information such as the user id for processing in subsequent routes. Again, you can test your implementation by using Postman: open a tab where you login a user via the login route, copy the returned token and open a new tab calling the gallery route passing the token as ‘`Authorization`’ header.

Exercise 3 – Requesting User Gallery

Complete the module ‘`gallery.js`’, which should implement retrieving a user specific gallery: users can own the same or different images (defined by table ‘`users_images`’) and this route should return all images for the currently logged in user. The images should be returned in JSON format, exactly like in EX4.4. Test and debug your implementation using Postman.

Exercise 4 – Gallery Client Side

Place ‘`nodejs_gallery_client`’ into your local webserver’s document root folder. This folder contains a modified client side gallery application based on EX4.4. It contains a login form, which should be used to login a user via the previously implemented login route. Your tasks are the following:

- ‘`site.js`’: Complete the function ‘`login`’ using POST to retrieve a JSON containing ‘`first_name`’, ‘`last_name`’ and ‘`token`’, which should be saved as a cookie – for this you can use the function ‘`createCookie`’. This cookie is used by ‘`init`’ in order to start loading galleries. Therefore, ‘`init`’ should be called after a successful login in order to begin loading a user specific gallery.
- ‘`JsonGallery.js`’: Fully implement the gallery’s ‘`getImages`’ function, which should handle retrieving the image list from your implemented gallery route (Hint: Remember to set the ‘`Authorization`’ header if you implemented authentication verification this way). Appropriately resolve/reject the created promise, passing the result JSON to the next function in the chain (‘`loadImages`’). After typing in a valid email/password you should now see and be able to browse the user specific gallery.

Exercise 5 – Changing Image Descriptions (Server & Client)

Provide logged in users with the functionality to change image descriptions. Start by implementing the PATCH image route provided by the server side module ‘`image.js`’. You should pass two parameters to this route: image id and description. The image id should be passed via the URL (‘`req.params.id`’), while the data using the body (‘`req.body.description`’) in JSON format (Hint: When using body and JSON, you must include ‘`"Content-Type": "application/json"`’ within the headers). Be sure to only allow users to change image descriptions for their own images. Finally, implement the client side function ‘`updateDescription`’ of ‘`JsonGallery.js`’, which should initiate the update process using the before implemented image route. Upon successfully changing a description in the database, also change the original description (‘`alt`’) of corresponding thumbnail in order to be able to see the effects without reloading the page. You should now have a fully functional Node.js gallery, implemented as a RESTful API.