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://node-postgres.com/

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

Exercise 1 – Node.js Basic Concepts

a) Explain the relation between JavaScript and Node.js.
b) Why is it a golden rule of Node.js not to block the event queue?
c) What is Express? What does it offer and what is it used for?
d) Explain the concept of Routes in Express.
e) What is the purpose of `app.all()`?

Exercise 2 – Simple Node.js Gallery

Write a server application using Node.js and Express with the following functionalities:

- When */galleryJSON* is requested on your Node.js server, it replies with a dynamically created JSON file which has the same structure as the one in assignment 5.2.
- However, you now need to read the file *gallery.csv* and convert the data to the proper JSON output format. You should create corresponding JavaScript objects and serialize them to JSON to generate the JSON output. Be sure to set the proper content type header!

To test your implementation, create an html document *client.html*, which

- includes and calls the attached file *client.js*
- contains appropriate html elements with the ids referenced in client.js
    - a textarea for displaying the JSON response (as string)
    - a div as container for the dynamically created image elements
- Also make sure that the port specified in client.js (3000) is the same you configured in your node.js installation, otherwise change it accordingly.

Exercise 3 – Familiarizing with the Node.js Gallery API

In the following exercises you are going to set up a Node.js multiuser gallery which interacts with a database for retrieving the images, image information, and user information. Users can interact with the gallery through a gallery API.

a) Setup the enclosed Node.js gallery API:
   - Place the folder `nodejs_gallery_server` in 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.
   - Run '`npm install pg`' for installing the Postgres Node.js module if needed.

b) Open the project in your text editor/IDE and inspect `server.js`:
   - Explain and reason about the structure of this node.js server application.
   - Explain the purpose of the files in the routes folder.
   - Which modules are used and what are their purposes?
   - What is Cross-Origin Resource Sharing (CORS)? What is its purpose?

c) Start the server by opening a console window, navigating to `nodejs_gallery` and running '`npm start`'
   - How does this command work?
   - What is the role of npm and npm scripts in general?

d) Install and run 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). Issue a simple GET request to '`localhost:3000`'. Explain the output.

e) What changes would be required 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 such that it returns the 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.

Exercise 4 – Setting up the DB connection for the Node.js Gallery

You will now setup and initialize a PostgreSQL database for a gallery service and connect it to the Node.js application.

a) Install and run Postgres if you don't have it already (https://www.postgresql.org).
b) Import the enclosed file `db_import/createDB.sql` to create and populate a new database '`webtech20gallery`'. The easiest way to do that is to use the included console tool *psql* (https://www.postgresql.org/docs/13/app-psql.html), but you might also use *pgAdmin*.
c) Use the web-based tool *pgAdmin* to verify if the database has been created correctly and explore the structure and content of the database tables. How many users and images have been inserted by the import script?
d) Complete the module '`db.js`': finish '`initDb()`' by connecting to the database resolving/rejecting the Promise on success/failure. The database should be accessible by any server component, simply by requiring and calling the '`getDb()`' function (e.g. see '`gallery.js`').

Exercise 5 – Implementing the Node.js Gallery API

You are now going to implement a first functioning API for the image gallery. This comprises several components that are required so that its functionalities can be exposed to the users.

a) Complete the module '`profile.js`', so that it queries the database and returns a rudimental profile of the user (first and last name in an appropriate JSON format), given that the password provided by the parameter ':`pass`' is correct for the user identified by the parameter ':`email`'.
    - Valid email/password combinations can be determined by inspecting the database of '`createDB.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 response status codes depending on the query outcome:
        - `400`: an error occurred
        - `401`: login failed
        - `200`: login successful
    - Test and debug your implementation with a POST request in Postman such as: `localhost:3000/profile/sandy@nomail.com/`*`12345`*

b) Complete the module ‘`gallery.js`’, which should implement retrieving a user specific gallery as list of image identifiers.
   - Users can own the same or different images (defined by table ‘`users_images`’) and this route should return all image ids for the user whose email address is passed as parameter.
   - The gallery should be returned as list of image ids in the following JSON format: `{imageIds: [Id1, …, Idn]}`.
   - Test and debug your implementation using <u>GET</u> requests in Postman such as: `localhost:3000/gallery/diana@nomail.com`

c) Extend the module ‘`gallery.js`’, so that it implements retrieving a specific image for a specific user gallery given the image id.
   - The image should be returned like in the following JSON format:
     ```
     {"dataSmall": "img/beach_small.jpg",
     "dataBig": "img/beach.jpg",
     "description": "Goleta Beach near UCSB Campus"}.
     ```
   - Make sure that invalid requests (i.e., invalid combinations of email and image id) return an error.
   - Test and debug your implementation using <u>GET</u> requests in Postman such as: `localhost:3000/gallery/sandy@nomail.com/1`

d) Provide users with the functionality to change image descriptions by completing the module ‘`image.js`’.
   - You should pass image id and description using the body (e.g., ‘`req.body.description`’) in JSON format
   - Hint: When using body and JSON, you must include ‘`"Content-Type": "application/json"`’ within the headers).
   - Test and debug your implementation using PUT requests in Postman.