Assignment 7

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 1: All exercises must be solved using node.js without using additional libraries or frameworks (except the ones proposed).

Note 2: We are not going to write code running within a browser for this assignment. To test the node.js code, it will be enough to use the client, which is able to run HTTP requests and show the HTTP response data. We recommend two clients:

1. Postman, available at https://www.postman.com, which is a desktop application. It is free for basic usage, which is enough for this course.
2. RESTer, which is a browser plugin. It is available for both Chrome and Firefox (firefox version is at https://addons.mozilla.org/en-US/firefox/addon/rester/, chrome version can be found in the Chrome web store (https://chrome.google.com/webstore, please search for RESTer there).

Exercise 1 – Getting Started with Node.js

---

1. Install Node.js
2. Take `ex1/server.js`, set the content type for output as indicated, install dependencies (express), and launch the server
3. Check the default route by visiting http://localhost:3000
4. Add the route with a handler to process route parameter passed through the URL like http://localhost:3000/mytext. The server should return HTML containing the parameter value like "`This is a simple application receiving` *`mytext`*"
5. Check this route by visiting http://localhost:3000/mytext

Exercise 2 – Simple Node.js Product List API

1. Take `ex2/server.js`, reuse content type handling from ex.1, read the list of products from `products.json` to be available within the code. Check the contents of the product array by writing it to the console.
2. Implement the `products` route, which returns all product IDs in JSON format as an array: `{ productIds: [ Id1, …, Idn ] }`
3. Test the route by visiting http://localhost:3000/products
4. Implement the `product` route to return JSON for a specific product by its ID to be called as http://localhost:3000/product/*ID* .
5. Return the error status code (e.g., 400) if there is no product with this ID; do this also for other routes in the subsequent exercises.
6. Test the route by visiting, e.g., http://localhost:3000/product/sony24105

Exercise 3 – Database-Backed Product List API

You will now set up and initialize a PostgreSQL database for a product list service and connect it to the Node.js application.

1. Install and run PostgreSQL if you don't have it already (https://www.postgresql.org).
2. Import the enclosed file `db_import/createDB.sql` to create and populate a new database, '`webtech22products`'. The console script is also provided; please change the user name there if you do not want it to be owned by Postgres users. Use the web-based tool *pgAdmin* to verify if the database has been created correctly and explore the structure and content of the database table.
3. Take `ex3/pool.js` and complete the module to set up the connection pool for your database. The connection parameters can be taken from `config.json`, change it to match your environment; they can also be hardcoded. The pool should be accessible from any server route handler simply by requiring this module and assigning the result to the pool variable: `const pool = require('./pool.js');` (see ex3/server.js)

4. Take `ex3/server.js`, start with the handlers from ex2, and change them to get the data from the database pool. Test the code by going to `products` and `product` URLs.
5. Implement the additional route handler to modify the description for a specific product. It should be called as a PUT request with
   a. product ID as a route parameter:

      PUT http://localhost:3000/product/sony24105

   b. a new value for a description as part of the JSON transferred as a request body.

   Note: to parse requests with JSON bodies, it is necessary to add the following to the server.js (this code is already in a provided stub, but pay attention to it anyway)

```
let bodyParser = require('body-parser');
app.use(bodyParser.json()); // support json encoded bodies
```

   It is also necessary to set the Content-Type for your requests to application/json in Postman or RESTer.

Exercise 4 – Product List API with Session Authentication

---

1. Import the enclosed file `db_import/createDB.sql` to create and populate a new database '`webtech22sessions`'. It includes the same `products` table as for ex3 and the `users` table with `login` and `password` fields and two users: admin/admin and user/user

2. Reuse `pool.js` and (partially) `server.js` from ex3, change `config.json` to match your environment

3. In the `server.js`, add session support as follows:
   a. Add the express-session (support for Express sessions) and connect-pg-simple (support for Postgres-based sessions) to the list of required modules at top of the server.js as follows

```
const session = require('express-session');
const pgSession = require('connect-pg-simple')(session);
```

      You also need to install both modules by means of npm.

b. Initialize the session support as follows

```
app.use(session( session-object ));
```

The *session-object* is a JavaScript object, which has to include, among others:

- The `store` attribute, which defines the object describing the session store. In our case, this object has to be an instance of the `pgSession` class, which indicates that the sessions have to be stored in the database:

  ```
  store: new pgSession(store-object),
  ```

  The *store-object* is a JavaScript object that has to include
  - The `pool` attribute which has to be initialized by the database pool such as was defined for exercise 3:
    ```
    pool: myPool // the name of the database pool
    ```
  - the `tableName` attribute, which has to be set to the name of the table used to store sessions:
    ```
    tableName: 'sessions'
    ```
  - the `createTableIfMissing` attribute, which indicates that if this table does not exist, it should be automatically created in the database:
    ```
    createTableIfMissing: true
    ```

- The `cookie` attribute defines the object to describe the cookie properties. For us, it is enough to specify the expiration time for the cookie by setting the `maxAge` attribute of this object:

  ```
  cookie: { maxAge: 1000 * 60 * 60 // 1
        hour
  }
  ```

- The `secret` attribute should contain the secret key for generating sessions; for our purpose, it is enough for this attribute to contain just "secret":

  ```
  secret: "secret"
  ```

Further changes to be made in `server.js` are described later.

4. Take `ex4/login.js` and implement a handler for the login route (a POST request handler). Within the handler for this route, you need to do the following:

   a. Get the user name and password from the request body, e.g., as the following JSON object: `{"user": "username", "pass": "password"}`,

   b. Check If the user with this password exists in a database; if so, handle the successful login by creating a new Express session. Such a session is available to every route as a session attribute of the request object (`req.session`). To create a session, its `isAuth` attribute has to be set to true. The user name can also be stored in a session as its `username` attribute.

   c. After creating the session, you return the user name as part of the JSON output to the client, together with a "login success" message. The session machinery in Express sets the cookie in a browser, which is transferred back to the server with every subsequent request until the session is destroyed.

   d. If the user does not exist, return the error status code to the client with a "login failed" message

   Test your route by issuing a POST request to http://localhost:3000/login and supplying admin/admin as user name and password within the JSON to be transferred as the request body. Check if the cookie is set in the browser.

5. Take `ex4/check_auth.js` and implement the module which exports a middleware function that is called before proceeding with any of the protected routes. Within this function, you should
   a. Check if the session exists by checking if `isAuth` attribute for the `session` attribute of the `req` object is set to true

   b. If the session exists, call the next middleware function (in our case, the protected route handler): `next();`

   c. If the session does not exist, return the error status code indicating the authentication failure

6. Take `ex4/server.js` again, and
   a. modify the protected route handlers to include the middleware function exported from `check_auth.js` module in the route chain (by passing the name of the function as the second parameter of `get()/post()/put()` calls, after the route path). The `check_auth.js` module has to be made required in the `server.js`, it is already done in a provided stub.

   b. Implement the `/logout` route (e.g. GET) with a handler that destroys the session object by calling `req.session.destroy()`, and checks if the session still exists by checking if `req.session` is defined; it should return the success code if it is so, and the error code otherwise.

7. Test your solution by
   a. Initializing the session by accessing the login route with a proper user name and password as part of the POST request body

   b. Accessing the protected routes, e.g., http://localhost:3000/products, these access attempts should succeed

   c. Accessing http://localhost:3000/logout

   d. Again, accessing the protected routes, these attempts should fail

   e. Checking if the session is persistent by logging in, stopping and restarting the server, and issuing a call for the protected route; it should work.