Assignment 7

Recommended readings:

- Lecture slides as starting literature
- Links within slides for details
- https://nodejs.org/en/docs/
- https://expressjs.com/en/guide/routing.html
- https://jwt.io/introduction/ and JWT Tutorial

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 Authentication via JSON Web Token

Answer following questions:

a) What is the purpose of route handlers in `express.js`?
b) What are middleware functions and how can they be used to change the behavior of a given route handler? How can you invoke the next callback of a route handler's middleware function chain?
c) What are JSON Web Tokens and how can they be used to authorize users?

Setup the enclosed Node.js gallery API:

- Prerequisites to install if you haven't done so for previous exercises
  - PostgreSQL with imported gallery database ('`createDB.sql`')
  - Node.js
  - Postman for easily issuing HTTP requests and testing the gallery API
- 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.
- Verify that everything is working via '`npm start`'. Expected output:

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

  Hint: This version of node.js gallery uses the '`nodemon`' package for monitoring any changes to server-side source files and re-running the server if files have been modified. Therefore, the server does not have to be manually restarted during development. For further information, please refer to: https://www.npmjs.com/package/nodemon.

Open the source code in `nodejs_gallery_server` in an editor/IDE of your choice and take a look the structure of the application. Identify changes in the application (especially in the route handlers '`image.js`'/'`gallery.js`') and reason about them.

Extend the provided server source code with a proper authentication functionality using the '`jsonwebtoken`' package:

- Alter '`login.js`' to accept user login information (email/password) in a more secure manner, i.e. by passing the information via the request body in JSON format (please note that this approach merely removes login credentials from plain sight, i.e. the URI, hence, still is not very secure, since HTTP transmitted data is not encrypted). Hint: Do not use Postman's built-in Authorization Tab for this task as it encodes base64 strings for login data, which will unnecessarily complicate login handling in server and later also client.
- Upon successful user login create a new JSON Web Token (JWT), which should be returned along with the response JSON to the client. The token is created with a simple JWT key as well as an expiration time, which are both defined in '`config.json`'. Be sure to include the user ID in the token payload data for being able to identify the authenticated user when handling authorization later. Finally, a logged in client should receive first-, lastname and token as a response. Test your implemented route (e.g. `localhost:3000/login/`) with Postman by issuing a POST request with appropriate credentials (for valid logins inspect database) in a JSON body – make sure to set the '`Content-Type: application/json`' header.
- Implement logged in user token authorization for all restricted routes in '`routes/image.js`' as well as '`routes/gallery.js`' by completing '`check_auth.js`'. This module exports a middleware function that is called before proceeding with any of the protected routes. It should receive a token – typically passed via the '`Authorization`' header (`req.headers.authorization`) – and verify if it is still valid, before allowing the request to proceed. Hint: Since '`req`' is accessible by the middleware function in '`check_auth.js`', you can use this object in order to store user information such as the user id for processing in subsequent functions. 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 in another tab test a different rout (e.g. `localhost:3000/gallery/`) passing the token as '`Authorization`' header (Headers Tab).

Exercise 2 – Node.js Gallery Client Side: Authorization

Place '`nodejs_gallery_client`' into your local webserver's document root folder (e.g. '`htdocs`' in XAMPP). This folder contains a modified client side gallery application based on EX5.5. The '`index.html`' includes a login form, which should be used to login a user via the previously implemented login route. Your tasks are the following:

a) '`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.

b) ‘`JsonGallery.js`’: Fully implement the gallery’s ‘`getImages`’ function, which should handle retrieving the image list from gallery route (Hint: Remember to set the ‘`Authorization`’ header if you implemented user 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 3 – Node.js Gallery Client Side: Changing Image Descriptions

Provide logged in users with the functionality to change image descriptions. Start by taking a look at the GET and PUT routes defined in ‘`image.js`’. Your task is to implement the client-side function ‘`updateDescription`’ of ‘`JsonGallery.js`’, which should first request the database entry for the currently maximized video issuing a GET request. Next the retrieved result should be altered based on the currently entered image description in the ‘`.desc`’ input field. The altered JSON should then be passed to the PUT route of the handler, which after successful authorization should update the database entry of the image. Finally, after finishing the database update, also change the original description of corresponding thumbnail (‘`alt`’ text) in order to correctly update the user indication for a changed image description (changed description: red, unchanged description: black). You should now have a fully functional Node.js gallery, implemented as a RESTful API.

Exercise 4 – RESTful Library Service 1

Answer following questions:

a) What are RESTful Services or RESTful APIs?
b) What is the notion of ‘Uniform Interface’ in a REST Service?
c) What is the difference between HTTP methods GET, POST, PUT, PATCH and DELETE? How should they be used?

Implement a simple RESTful library service using ‘`express`’, which offers users to rent and return books via appropriate requests. The library of available books is defined in ‘`book_library.json`’. You only need to implement the service, i.e. the server side of the application and test your implementation with Postman. Your tasks are the following:

- Load ‘`book_library.json`’ upon starting the service, keeping the library in memory.
- Implement a route handlers for
  - retrieving all books (e.g. ‘`/books`’)
  - retrieving a single book by ISBN number (e.g. ‘`/books/:isbn`’)
  - renting a single book by ISBN (e.g. ‘`/books/:isbn`’) setting ‘`rented=true`’

- returning a single book by ISBN (e.g. '`/books/:isbn`') setting '`rented=false`'

Be sure to follow the REST paradigm for this exercise using appropriate HTTP methods for all defined routes.

Exercise 5 – RESTful Library Service 2

Extend your library service by adding routes for creating and deleting books, again following the REST paradigm. Additionally, provide a key-specific search functionality, e.g. using the '`/books`' route, capable of finding books by partially matching ISBN, authors, title or category. Use query string parameters for such a request, which should return a collection of JSON entries – e.g. searching for a '`title`' containing '`node`' yields following list:

```
[
    {
        "title": "Node.js in Action",
        "isbn": "1617290572",
        "pageCount": 300,
        "publishedDate": {
            "$date": "2013-10-15T00:00:00.000-0700"
        },
        "thumbnailUrl": "https://s3.amazonaws.com/AKIAJC5RLADLUMVRPFDQ.book-thumb-
images/cantelon.jpg",
        "shortDescription": "Node.js in Action is …",
        "longDescription": "JavaScript on the …",
        "status": "PUBLISH",
        "authors": [
            "Mike Cantelon",
            "Marc Harter",
            "T.J. Holowaychuk",
            "",
            "Nathan Rajlich"
        ],
        "categories": [
            "Web Development"
        ],
        "rented": false
    },
    {
        "title": "Node.js in Practice",
        "isbn": "1617290939",
        "pageCount": 0,
        "thumbnailUrl": "https://s3.amazonaws.com/AKIAJC5RLADLUMVRPFDQ.book-thumb-
images/templier2.jpg",
        "status": "MEAP",
        "authors": [
            "Alex Young",
            "Marc Harter"
        ],
        "categories": [],
        "rented": false
    },
    {
        "title": "Getting MEAN with Mongo, Express, Angular, and Node",
        "isbn": "1617292036",
        "pageCount": 0,
        "thumbnailUrl": "https://s3.amazonaws.com/AKIAJC5RLADLUMVRPFDQ.book-thumb-
images/sholmes.jpg",
        "status": "MEAP",
        "authors": [
            "Simon Holmes"
        ],
        "categories": [],
        "rented": false
    }
]
```