Assignment 7

So far, Assignments were mainly introductions to JavaScript and related technologies. With Assignment 7, however, we move server-based implementations one step further towards production-level code. In Assignment 7, we look into a possible way of deploying real-world applications. Besides, user authentication was not yet covered. With JSON Web Tokens a secure state-of-the-art solution for authentication is introduced. The last missing component are REST APIs. With REST, structured and intuitive application interfaces are introduced, increasing the overall quality of server applications.

The provided materials contain two project templates. The first (folder *ex1-4*) contains an adapted version of the JSON image gallery, with pre-implemented code for DB operations. The second (folder *ex-5-6*) is a REST-based application template that should implement a library. It keeps track of a list of books and monitors whether they available are or not.

Recommended readings:

- Lecture slides as starting literature
- Links within slides for details
- Deploying to production: MDN web docs
- Heroku, aample hosting provider: Heroku.com
- https://jwt.io/introduction/ and JWT Tutorial
- What is REST: https://restfulapi.net/

Exercise 1 – Node.js Gallery Authentication via JSON Web Token

Answer following questions:

a) 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?
b) What are JSON Web Tokens and how can they be used to authorize users?

Note: Switch back to a local development environment rather than testing your app on Heroku. Therefore, you can use the DB created for Assignment 6.

Setup the Node.js gallery API:

- Prerequisites to install if you haven't done so for previous exercises:
  - PostgreSQL with imported gallery database ('`createDB.sql`' from Assignment 6)
  - Node.js
  - Postman, or a similar tool for easily issuing HTTP requests and testing the gallery API
- Place `nodejs_gallery_server` (from folder *ex1-2*) 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.

  <u>Hint</u>: 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. This folder contains a modified client side gallery application based on Exercise 3.4. 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 – RESTful Library Service I

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`' (from folder *ex3-4*). 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 4 – RESTful Library Service II

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
    }]
```

Exercise 5 – Deploy your Application

Browse the Heroku documentation and answer the following questions:

- What is Heroku used for?
- What is a Dyno and how can Dynos help to scale your app?
- How can Databases be supported by Heroku?
- What is the standard way of deploying your app on Heroku?
- How can you see the console output of a deployed application?

The goal of this exercise is to deploy the provided server application template from folder *ex1-2*. This can be done by completing the following step-by-step instructions:

- Sign up for a Heroku free account and create a new application. Choose *gallery-server* as app name and Europe as region. You are forwarded to the dashboard of your newly created application on Heroku. Take a look at the created application name, you will need it later when configuring your DB.
- Install the Heroku CLI and initialize a Git repository for the provided *nodejs_gallery_server* application. Be careful to not commit the *node_modules/* folder.
- Log in with the installed Heroku command-line tool.
- Follow the instructions from your application dashboard to deploy your app. Therefore, add the provided remote to your local git repository and push it to Heroku.

- **You're done!** At least with the first step: your first node.js app is deployed and reachable via a link shown on the command-line.
- Look at your app's logfile (*heroku logs* on command-line), the output contains something like the following:

```
2020-11-28T06:16:14.908673+00:00 app[web.1]: Error: connect ECONNREFUSED 127.0.0.1:5432
2020-11-28T06:16:14.908675+00:00 app[web.1]: at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1144:16)
2020-11-28T06:16:14.908998+00:00 app[web.1]: Failed to connect to DB!
```

Right, we forgot our DB! Let's go on configuring a free postgres instance.

- Visit data.heroku.com and configure *Heroku Postgres* for your application. Therefore, select the free plan and configure your app to provision.
- You are forwarded to your application dashboard. Your DB is now configured and ready to use.
- Visit the *Resources* tab on your application dashboard and click on *Heroku Postgres* → *Settings* → *View Credentials* to get information about how to connect to your database.
- Use your postgresql client to execute the install script on the newly created database.
  Hint: Use the new script provided with Assignment 7. The script from Assignment 6 also creates users, which is not possible in the Heroku environment.
- Open the *config.json* of the *nodejs_gallery_server*. Edit the settings for the database host, user, password, and DB. Commit the changes and push them to Heroku.
- Again, view the application logfiles.
- Verify that your application is working as expected by issuing HTTP requests to your application (e.g. by using Postman).

Great! You just deployed your first application including Database to Heroku. This means you uplifted your application from local development to a production environment. However, keep in mind that user authentication is still missing, and REST principles are not yet applied.

You can stop your application with the following command:
```
heroku ps:scale web=0
```