CAB230 Assignment 3, 2023
Server-Side

Introduction

In the 2nd assignment, you developed a React application that allowed users to view information about movies and the people who create them. The data was accessed from a REST API that we provided you. In this assignment, your task is to create the server hosting that same API (with a couple of small additions.)

This assignment builds upon the lecture and practical material covered in the second half of CAB230. In particular, you must use the following technologies:

- Node
- Express
- MySQL (or MariaDB)
- Swagger
- Knex
- JSON Web Tokens

No alternatives to these core technologies are permitted. An essential part of this assignment is to deploy your Express application to the provided virtual machine that we have allocated to you. Information on this was provided in Week 11.

Although this assignment shares a lot in common with the second assignment, insofar as we are using the same movies dataset and working with a common API, your grade level in assignment two will have no bearing on your grade level for assignment three – so treat it as a clean slate!

The Data

The movies database is provided as a SQL dump in the assignment_data.zip file. You should import this database following a similar process to the "World Cities" database that we have been working on in the practicals. Once the dataset has been successfully imported, you should not touch it at all (except to add a table or tables for storing user details.)

The database consists of the following tables:

- **basics** – containing information about each film
- **crew** – (not needed for this assignment)
- **names** – contains information about people, used for the /people/{id} endpoint
- **principals** – containing information about 10 of the main people that worked on the film
- **ratings** – containing ratings from IMDB, Rotten Tomatoes and Metacritic
- **ratingsold** – (not needed for this assignment)

Open up the database in MySQL Workbench and look around to get a feel for the fields in the database and the types of values to be found in each. Note that some of these fields contain null values, usually because some information was missing or is not relevant. Note that we have not provided a table for storing user data – you will need to create this yourself. Additionally, note that

advanced SQL knowledge is not required for any part of this assignment – it is not necessary to use JOINs, although you are permitted to use them.

The REST API

The API you are required to implement is identical to the one hosted at http://sefdb02.qut.edu.au:3001/, except with two additional endpoints that you also need to implement. These endpoints are for storing and retrieving profile information about the users.

Rate-limiting

It is not necessary for your assignment to support rate-limiting, or to produce 429 errors. If you do wish to implement rate-limiting, you can use express-rate-limit, but be warned – during automated testing we will hit your API with many requests very quickly.

The profile routes

You are required to, provide, in addition to the routes in the Assignment 2 API, two additional routes: GET /user/{email}/profile and PUT /user/{email}/profile. You need to implement these routes and add documentation to them to the Swagger file.

GET /user/{email}/profile

This route returns a user's profile information as a JSON object. There are two different types of output presented here – one if the request is authorised and if the bearer token presented was issued to the user whose profile is being retrieved, and the other if it is not. An unauthorised request (without an 'Authorized:' header) or a request from a different user will receive an object like this:

200 OK

```
{
  "email": "mike@gmail.com",
  "firstName": "Michael",
  "lastName": "Jordan"
}
```

The same request, but with a valid JWT bearer token belonging to the profile's owner, will receive an object like this, with additional fields for date of birth and address:

200 OK

```
{
  "email": "mike@gmail.com",
  "firstName": "Michael",
  "lastName": "Jordan",
  "dob": "1963-02-17",
  "address": "123 Fake Street, Springfield"
}
```

The above two examples are of successful requests, which come with the HTTP status code of 200. However, if there is a problem with the JWT token, your server will return one of the following responses with a status code of 401 Unauthorized.

If the JWT token has expired:

401 Unauthorized

```
{
  "error": true,
  "message": "JWT token has expired"
}
```

If the JWT token failed verification:

401 Unauthorized

```
{
  "error": true,
  "message": "Invalid JWT token"
}
```

If there was an 'Authorization:' header, but it did not contain 'Bearer ' followed by the JWT:

401 Unauthorized

```
{
  "error": true,
  "message": "Authorization header is malformed"
}
```

If {email} corresponds to a non-existent user, the following response will be returned with a status code of 404 Not Found:

404 Not Found

```
{
  "error": true,
  "message": "User not found"
}
```

Note that a newly created user will not have any of these fields filled in. Your server will return null for any fields that have not been provided, e.g.:

200 OK

```
{
  "email": "notmike@gmail.com",
  "firstName": null,
  "lastName": null,
  "dob": null,
  "address": null
}
```

PUT /user/{email}/profile

This is used to provide profile information. The request will contain a body with the application/json content-type containing profile fields in the same format that they are returned with GET /user/{email}/profile (except without the email address):

```
{
  "firstName": "Michael",
  "lastName": "Jordan",
  "dob": "1963-02-17",
  "address": "123 Fake Street, Springfield"
}
```

Users can only change their own profile information. In other words, if you are logged in as mike@gmail.com, you can change the profile of mike@gmail.com, but not any other user.

If you successfully update a profile, the response will be an object containing the updated profile:

200 OK

```
{
  "email": "mike@gmail.com",
  "firstName": "Michael",
  "lastName": "Jordan",
  "dob": "1963-02-17",
  "address": "123 Fake Street, Springfield"
}
```

If the user is logged in with the wrong email (that is, the JWT is provided and is valid, but the credentials do not belong to the user whose profile the user is attempting to modify) your server will return the following:

403 Forbidden

```
{
  "error": true,
  "message": "Forbidden"
}
```

If there is no Authorization: header:

401 Unauthorized

```
{
  "error": true,
  "message": "Authorization header ('Bearer token') not found"
}
```

If the JWT token has expired:

401 Unauthorized

```
{
  "error": true,
  "message": "JWT token has expired"
}
```

If the JWT token failed verification:

401 Unauthorized

```
{
  "error": true,
  "message": "Invalid JWT token"
}
```

If there was an 'Authorization:' header, but it did not contain 'Bearer ' followed by the JWT:

401 Unauthorized

```
{
  "error": true,
  "message": "Authorization header is malformed"
}
```

If {email} corresponds to a non-existent user, the following response will be returned with a status code of 404 Not Found:

404 Not Found

```
{
  "error": true,
  "message": "User not found"
}
```

If the submitted object does not contain all of the fields:

400 Bad Request

```
{
  "error": true,
  "message": "Request body incomplete: firstName, lastName, dob and address
are required"
}
```

If any of the fields are not strings:

400 Bad Request

```
{
  "error": true,
  "message": "Request body invalid: firstName, lastName, dob and address
must be strings only"
}
```

If the date of birth is not a valid YYYY-MM-DD date (e.g. no April 31 or February 30, or February 29 on a non-leap year):

400 Bad Request

```
{
  "error": true,
  "message": "Invalid input: dob must be a real date in format YYYY-MM-DD"
}
```

The HTTP method used for this endpoint is PUT. PUT is a HTTP method similar to POST, though with different semantics. You can install express middleware for a PUT method with .put(), just as you use .post() to set up POST routes.

Your task is to implement both of these routes on the server, and also to modify the Swagger doc to document these routes, including all of the error codes.

Grade standards

We expect that you will follow a professional approach in the architecture and construction of the server. In particular:

- The routes and the overall structure are professional – logical and uncluttered, with appropriate use of specific routers mounted on the application
- There is appropriate use of middleware for managing components such as database connectivity and security
- There is appropriate error handling and the responses match those listed in the Swagger documentation (or in this specification, in the case of the profile routes)
- That the Swagger documentation has been amended to include both of the new profile routes and that the documentation matches the specification for these routes, as is documented in this file
- The application successfully serves the Swagger docs on the home page route
- The application is successfully deployed to a QUT VM using HTTPS with a self-signed certificate
- There is appropriate attention to application security

Broadly speaking, the grade standards for this assignment correspond to the feature levels laid out below. Similarly to assignment two, the grade levels below assume that the features have been implemented competently. If the implementation is substandard, the marks awarded will be reduced (perhaps substantially).

[Grade of 4 level]: Successful deployment to a QUT VM of an Express-based REST API which supports the basic endpoints (/movies/search, /movies/data/{imdbID} and /people/{id}), though authentication may be missing or with significant issues. The Swagger docs may not be served, or may simply be unaltered from the file we provide. There may be significant gaps in the security requirements.

[Grade of 5 level]: All of the routes except /user/refresh and /user/logout are implemented, at least to a basic level. Registration, login and JWT token handling must be attempted – although there may still be some minor issues at this level. A reasonable attempt must have been made at the profile routes, although we would expect significant limitations. The Swagger docs are served, although you may not have added the entries for the new profile routes.

[Grade of 6 or 7 level]: All of the routes have been successfully implemented. The profile routes must work properly in both authorised and unauthorised modes at the 7 level. The distinction between the 6 and 7 level is based on how well the API implements error responses, successful use of middleware security, database connectivity and the correctness of the Swagger docs. There is no one requirement that means the difference between a 6 or a 7 – these requirements are just to give you an idea of what we are expecting.

Submission

You are required to both submit your assignment code to Canvas (remove 'node_modules' and zip up the entire project, just as you did for Assignment 2) and have your assignment deployed on the provided virtual machine. More details on the virtual machine were made available in Week 11 – watch the Deployment videos and the lecture recording.

Unlike Assignment 2, no report needs to be submitted this time – the only documentation you need to write is the modified Swagger file.

Acknowledgements

The data is sourced from OMDb and is licensed under CC BY-NC 4.0.