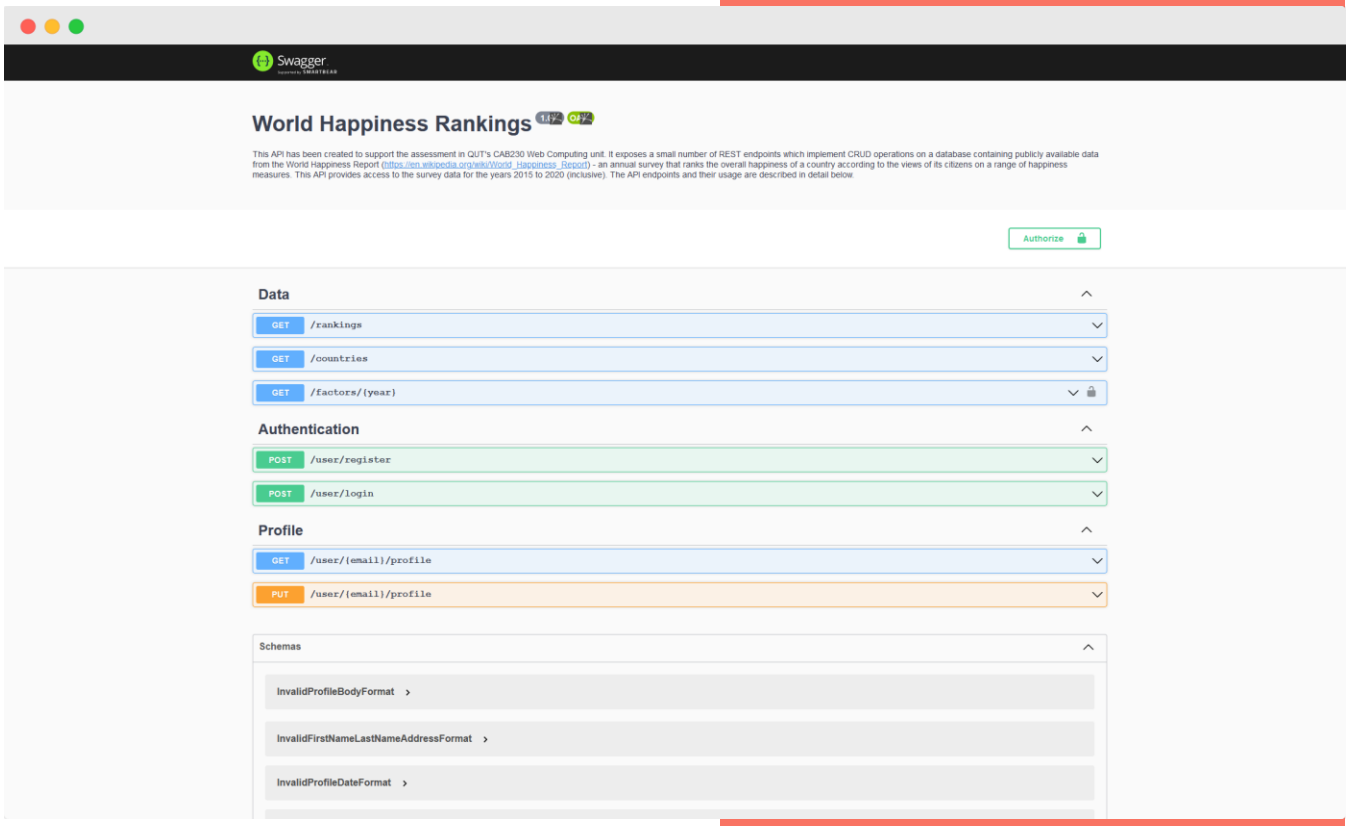
2021

CAB230 Assignment 2 Server Side



Benjamin Rogers

N10212868

5/4/2021

Contents

Introduction:

Purpose and description

The purpose of this API was to search as a backend for the client-side happiness application. The dataset is based on publicly available data released as part of the World Happiness Report and includes ranks for each country, their assigned happiness score, and a range of related factors. The API provides routes for user registration and login that replicate those used in the client-side application. There are also two additional routes for updating and retrieving user profile information. The API also has Swagger documentation to explain each of the routes, how to use them, and expected error messages that might get returned.

Completeness and limitations

All the required functionality was implemented. All routes were implemented. The data routes are fully functional with proper filtering and authentication for the factors route. Login and registration work as expected and handle the JWT appropriately. The profile routes also work properly in both authorised and unauthorised modes. There is also middleware security, database connectivity and deployment of Swagger documentation. Each of the routes handles errors gracefully and errors are returned as per the Swagger documentation. Self-signed HTTPS was also successfully set up alongside TLS.

Routes:

All Routes are fully functional.

/rankings

Returns a list of countries and their happiness rank for the years 2015 to 2020. The list is arranged by year, in descending order. The list can optionally be filtered by year and/or country name using query parameters.

/countries

Returns a list of all surveyed countries, ordered alphabetically.

/factors/year

Returns a list of countries and their associated happiness factor scores for the specified year. The path parameter (year) is required. The number of returned results can be limited by the optional limit query parameter. A result for a single country can be obtained via the optional country query parameter. This route also requires the user to be authenticated - a valid JWT token must be sent in the header of the request. A JWT token can be obtained by logging in.

/user/register

Creates a new user account. A request body containing the user to be registered must be sent.

/user/login

Log in to an existing user account. A request body containing the user credentials must be sent.

/user/{email}/profile

Users can both retrieve and update their profile information using this route. Profile information includes first name, last name, dob and address. This route is composed of two components which are the public and authenticated sides. Public fields are email, first name and last name. Authenticated fields are dob and address, which can only be accessed by the profile owner.

Additional Modules Used:

Yaml.js

Standalone JavaScript YAML 1.2 Parser & Encoder. Works under node.js and all major browsers. Also brings command line YAML/JSON conversion tools. This was used so that the swagger documentation could be written in YAML instead of JSON.
https://www.npmjs.com/package/yamljs

Moment.js

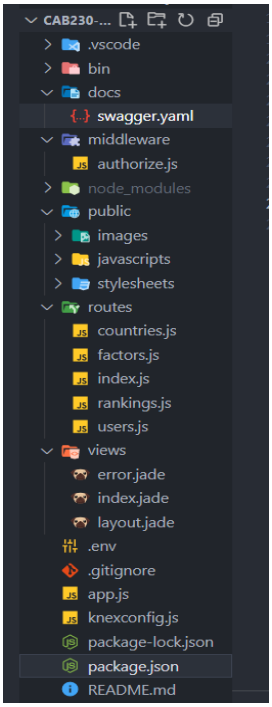
A JavaScript date library for parsing, validating, manipulating, and formatting dates.
https://www.npmjs.com/package/moment

Technical Description:

Architecture

Structure

The organisation of the application is based on express generator with folders for bin, public, routes, and views. The additional folders I added were middleware and docs. The middleware folder houses a middleware function called authorize that is used for routes that require a user to be logged in. The docs folder contains the YAML file for the Swagger documentation so that it is neatly organised. In the routes folder, I replaced kept the index.js file for serving the Swagger docs at the "/" but I also separated the other routes into their own files. These include countries.js for "/countries", factors.js for "/factors", rankings.js for "/rankings" and users.js for "/user". The users.js file contains the POST routes for login, registration, as well as the GET route for profile and the PUT route for profile. I have also got an app.js in the root directory as per the express generator setup, and this specifies the application itself. Additionally, I have a .env file to store information such as port number and secret key for better security. Lastly, I have knexconfig.js in the root directory which contains configuration for the MySQL database.



As mentioned, the middleware folder houses an authorize function which is exported for the entire application to use. This authorize function is what is used in routes that require authorisation to block or grant access. It also checks if the JWT is malformed, invalid, expired or not found and returns an appropriate message for each. This authorize function is used in the factors route and the PUT route for profile. It was not used in the GET profile route because an unauthorised user needed to still be able to access other user profile.

```
const jwt = require("jsonwebtoken");

module.exports.authorize = (req, res, next) => {
  const authorization = req.headers.authorization;
  let token = null;

  // Retrieve token
  if (authorization) {
    // Check if malformed
    if (authorization.split(" ")[0] != "Bearer") {
      res
        .status(401)
        .json({ error: true, message: "Authorization header is malformed" });
      return;
    }
    // If not malformed, stored it
    if (authorization.split(" ").length === 2) {
      token = authorization.split(" ")[1];
    }
  }

  // Verify JWT and check expiration date
  try {
    const decoded = jwt.verify(token, process.env.SECRET_KEY);

    if (decoded.exp < Date.now()) {
      res.status(401).json({ error: true, message: "JWT token has expired" });
      return;
    }
    next();
  } catch (e) {
    if (e.message === "jwt must be provided") {
      res.status(401).json({
        error: true,
        message: "Authorization header ('Bearer token') not found",
      });
    } else {
      res.status(401).json({ error: true, message: "Invalid JWT token" });
    }
  }
};
```

Security

The OWASP security principles were carefully considering when these security features were implemented.

1. Injection:

   This has to do with vulnerabilities in SQL, queries and is prevalent, particular in legacy code. For this application, Knex.js was used for all the SQL queries across the application. Knex provides some additional security by preventing the need for raw SQL in the code. Instead, Knex uses query builder notation, and this prevents SQL injection attacks (OWASP, 2021). This prevents issues such as the risk of data loss, corruption, disclosure to unauthorised parties, loss of accountability and denial of access (OWASP, 2021).

2. Broken Authentication

   Attackers could gain access to usernames and passwords easily if the authentication procedures are not well implemented. Session management is a big concern and JWT tokens should not last forever. JWT was used for managing logged in users across the app. Generating a JWT can be found at the bottom of the login route. The secretKey variable is used to produce the signature of the JWT token which is important for verifying the user. This secret key is kept private within the

```
queryUsers
  .then((users) => {
    if (users.length === 0) {
      res.status(401).json({ error: true, message: "User does not exist" });
      return;
    }
    const user = users[0];
    return bcrypt.compare(password, user.password_hash);
  })
  .then((match) => {
    if (!match) {
      res
        .status(401)
        .json({ error: true, message: "Incorrect email or password" });
      return;
    }
    const secretKey = process.env.SECRET_KEY;
    const expires_in = 60 * 60 * 24; // 1 Day
    const exp = Date.now() + expires_in * 1000;
    const token = jwt.sign({ email, exp }, secretKey);
    res.json({ token_type: "Bearer", token, expires_in });
    currentUserEmail = email;
  });
});
```

.env file as mentioned earlier. The expires_in and exp variables defined how long the token will remain valid from the time it is created. The email is also stored in the JWT which is used for verifying the logged in user when updating their profile information in the PUT route. These features prevent issues that could come up as result of broken authentication. Broken authentication could cause credential stuffing or application session timeout-related attacks (OWASP), 2021)

3. Sensitive Data Exposure

Rather than directly attacking crypto, attackers steal keys, execute man-in-the-middle attacks, or steal clear text data from the server, while in transit, or from the user's client. The most common example of this is not encrypting sensitive data. For storing passwords, a library called bycrpt was used to hash and compare passwords. When the password is hashed, a salt is used so that if an attacker got one of the passwords and other users have the same password, then the others are not at risk since each hashed password is unique. The saltRounds variable set to 10 will ensure that the attacker will not be able to decrypt the password in any reasonable amount of time. When logging in, bycrpt.compare() is used to check the passwords

4. XML External Entities (not relevant)

5. Broken Access Control

Access control weaknesses are common due to the lack of automated detection, and lack of effective functional testing by application developers. For this application, users are only granted access to the functions that they should be. Users are not able to update the profile of other users other than themselves. For the GET profile route, if a JWT token is sent with a request, then the email associated with the token must match the email in the URL to get the additional dob and address data. For the GET profile route, the email must also match to be able to update your own profile data.

6. Security Misconfiguration

Attackers will often attempt to exploit unpatched flaws or access default accounts, unused pages, unprotected files, and directories, etc to gain unauthorized access or knowledge of the system. One security configuration measure that I used was putting the secret key within a .env file so that if the source code is published online, the secret key is never kept within the source code itself. If this were to be used by people online, my .env file should not be uploaded but there would instead be a .env sample file where the user would place a secret key of their own in there.

7. Cross-Site Scripting (XSS)

```
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, "public")));
app.use(logger("common"));
app.use(helmet());
app.use(cors());
```

Cross-site scripting is a type of security vulnerability where malicious code is injected into a web page. One of the main concerns with XXS in a REST API is SQL injection however this is prevented using Knex as mentioned earlier. Using Helmet is also a good measure which was incorporated within this application. Helmet helps secure the Express app by setting various HTTP headers (StrongLoop, 2021). Some of these header's help mitigate cross-site scripting attacks, among other things. Helmet is essentially a collection of smaller middleware functions used by the application. Helmet was set in the app.js file.

8. Insecure Deserialization (not relevant)

9. Using Components with Known Vulnerabilities

This mainly deals with checking that external libraries that are used are the latest version to prevent vulnerabilities. Out of date components have been known to lead to major data breaches in the past. Flaws in any component can result in serious impact. All libraries and external modules used in the development of this app are the latest version.

10. Insufficient Logging and Monitoring

```
const helmet = require("helmet
```

```
app.use(logger("common"));
```

```
// Logger format
logger.token("req", (req, res) => JSON.stringify(req.headers));
logger.token("res", (req, res) => {
  const headers = {};
  res.getHeaderNames().map((h) => (headers[h] = res.getHeader(h)));
  return JSON.stringify(headers);
});
```

Attackers rely on the lack of monitoring and timely response to achieve their goals without being detected. Insufficient logging and monitoring can result in serious consequences if exploited. Morgan was used to manage logs for requests made to the server. Morgan is a middleware logger that logs information about HTTP requests and errors (DigitalOcean, 2021). The use of Morgan can also be seen in the app.js file. By default, Express generator includes a reference to Morgan. The logger format is also specified by "common" which is just one of Morgan's predefined formats (DigitalOcean, 2021). It is the standard Apache combined log output. Server administrators can take advantage of a tool like Morgan so that they can monitor activity that is happening on the server during periods that they might not be able to. Logging can even be configured to save to a file so that all the logs can be viewed later. If suspicious activity were to occur during the middle of the night for example, then a logger like Morgan will keep track of this.

The application was also deployed using HTTPS alongside TLS/SSL. HTTPS was set up very easily. To do so, 'http' was changed to 'https' in the server configuration within './bin/www'.

```
$ sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -
keyout /etc/ssl/private/node-selfsigned.key -out
/etc/ssl/certs/node-selfsigned.crt
```

TLS provides the application with encryption, authentication, and integrity. TSL/SSL is configured by using a combination of a public key and a private key. The SSL key is kept secret on the server which is used to encrypt content sent to the clients. The SSL certificate is publicly shared. The certificate is necessary for the website to have HTTPS encryption and helps prevent man-in-the-middle attacks, domain spoofing and other forms of attacks (OWASP, 2021). The self-signed key and certificate are created with OpenSSL using the command seen on the right. The key and credentials are read using the code here as well within './bin/www'. The port number is also set to 443 since this is the default for HTTPS. It is worth mentioning that the VM is behind a QUT firewall, so UFW was not considered.

```
var port = normalizePort(process.env.PORT || "443");
app.set("port", port);
```

```
var server = https.createServer(credentials, app);
```

Testing

All tests were passed and the results from the report can be seen below.

Test Report

Started: 2021-06-08 16:28:44

Suites (1)	Tests (301)
1 passed	301 passed
0 failed	0 failed
0 pending	0 pending

C:\Users\benro\OneDrive\Documents\GitHub\happinessapi-tests\integration.test.js 2.656s

countries > with no query parmater	*return a list of all countries*	passed	0.001s
countries > with no query parmater	*should return status text - OK*	passed	0s

Difficulties and Errors

One issue I had when developing the application was with date validation. Since the user can update their date of birth, there needed to be a way of checking if DOB was in the correct format and if it was before the current date. To do this I used the Moment.js library which provided methods such as isValid() and isAfter() which made checking these two conditions easy. I could easily specify the format of dob as 'YYYY-MM-DD'. I also made moment use strict parsing with it set to true s that it requires the format and input to match exactly, including delimiters.

```
let currentDate = moment();
let birthDate = moment(dob);

// Check format of dob
if (!moment(dob, "YYYY-MM-DD", true).isValid()) {
  console.log(dob);
  res.status(400).json({
    error: true,
    message: "Invalid input: dob must be a real date in format YYYY-MM-DD.",
  });
  return;
}

// Check if dob is out of bounds
if (birthDate.isAfter(currentDate)) {
  res.status(400).json({
    error: true,
    message: "Invalid input: dob must be a date in the past.",
  });
  return;
}
```

Another difficulty I had was working out how to make sure both authenticated and unauthenticated users can access the GET profiles route. Initially, I was trying the export the email from the authorise function when a user logs in. However, I decided that it made more sense to decode the email within the route as the authorise function is not used in the GET profile route. In the GET, there is a check to see if the authorization header is not undefined. If it is not, then this means there was a JWT sent with the request and the email associated with that token is then checked. If the token email matches the email in the URL, then the SQL query is updated to include dob and address, otherwise the SQL query includes just first name, last name, and email.

```
// If user logged in requests their own profile,
// they also get dob and address
if (req.headers.authorization !== undefined) {
  const authorization = req.headers.authorization;
  let token = null;
  if (authorization.split(" ").length === 2) {
    token = authorization.split(" ")[1];
  }
  const decoded = jwt.verify(token, process.env.SECRET_KEY);
  if (decoded.email === req.params.email) {
    query = req.db
      .from("users")
      .select("email", "firstName", "lastName", "dob", "address")
      .where("email", "=", email);
  }
}
```

Another design decision based on an issue I was having was the organisation of the authorize function. I needed it in two separate routes – factors and profile. Instead of repeating code, I put the function in its own file and exported it for the whole application to use.

Installation Guide:

Get Started

You need MySQL version 8.0.24, Node and npm to run this project.

From your command line, first clone this repo:

```
# Clone this repository
$ git clone https://github.com/ben04rogers/cab230assignment2.git

# Go into the repository
$ cd cab230assignment2
```

Then you can install the dependencies using NPM or Yarn:

Using NPM:

```
# Install dependencies
$ npm install
```

Replace the SECRET_KEY in the .env file with a secret key.

The data is available from the SQL dump file at https://drive.google.com/drive/u/1/folders/1ZUy0MKcLtuvOQ5b3NQWididxd2jU5Vw- and that will be necessary to create both the rankings, and users tables. These are the following fields on the database:

Table: rankings

Columns:

ID	int AI PK
rank	int
country	varchar(45)
score	decimal(4,3)
economy	decimal(4,3)
family	decimal(4,3)
health	decimal(4,3)
freedom	decimal(4,3)
generosity	decimal(4,3)
trust	decimal(4,3)
year	year

Table: users

Columns:

id	int AI PK
email	varchar(120)
firstName	varchar(50)
lastName	varchar(50)
dob	date
address	varchar(200)
password_hash	varchar(120)

To start the application

```
# Start express server
$ npm start
```

References

OSWASP (2021). OWASP Top Ten Web Application Security Risks | OWASP. Retrieved 11 June 2021, from https://owasp.org/www-project-top-ten/

StrongLoop (2021). Security Best Practices for Express in Production | StrongLoop. Retrieved 11 June 2021, from https://expressjs.com/en/advanced/best-practice-security.html

Digital Ocean. (2021). How To Use morgan in Your Express Project | DigitalOcean. Retrieved 11 June 2021, from https://www.digitalocean.com/community/tutorials/nodejs-getting-started-morgan

Ishan. (2021). Password Encryption — Hashing in Node Application. Retrieved 11 June 2021, from https://itnext.io/password-encryption-hashing-in-node-application-311a6f61cd65