# CAB230 Server side report

Queensland University of Technology
Samuel James Terbio Holden
n10110381

---

## Introduction

Below are what is expected for the server side, and my commented regarding completion or limitations

- The routes and the overall architecture are professional – logical and uncluttered with appropriate use of specific routers mounted on the application
  - Routers are used with path '/…' leading to the open endpoints or the swagger doc and '/search' directs to the secure endpoint by first checking the token validity before proceeding with the request
- There is appropriate use of middleware for managing components such as database connectivity and security
  - The server utilises helmet to direct users to only access the secure https, knex to connect and query to the SQL database, bcrypt to hash passwords to store in the database, and JWT to authenticate users
- There is appropriate error handling and responses match those in the API spec
  - Based on my tests any error responds with the same status codes however with some differing message responses
- The application is successfully deployed using HTTPS
  - The server is deployed using HTTPS
- There is an appropriate attention to application security.
  - Passwords are sent via HTTPS and the SSL making the password encrypted end to end and when the password is stored, the app will hash the password with a salt so that the database doesn't store plaintext. The login route will return a generated JWT that is checked for validity by ensuring the encrypted email stored is in the database and that it hasn't expired.
- The application successfully serves accurate Swagger docs on the home page route
  - Swagger is running on the home page route allowing for the user to test the endpoints. The secure search endpoint can also be used in the swagger document by supplying a JWT

## Technical description

The packages used follow those that are found in the tutorials.

Using the initial express app I have created 3 routes with the path reading “/…“, “/search“, “/“. The first path will respond to helper endpoints, offences, ages, areas, genders, and years, this will also respond to the login and register endpoints. The third path will proceed straight to the swagger docs.

I have also used https, bcrypt, jwt, and helmet which I will discuss in the following security section.

## Security

For security I have utilised https, bcrypt, jwt, and helmet. HTTPS allows for the use of the ssl to ensure the password will be encrypted during transmission from the client to the api and vice-versa. Using bcrypt allows for the use of hashing the users’ passwords to store in the database without revealing the password to anyone who may have gained access into the SQL database. JWT is used in order to create a token that can be sent to the user that they provide in order to gain access to the database of the crime statics. Helmet is used so that the all requests will redirect to https and stop http connections to ensure security and end to end encryption.

## Testing

| Test Case | Input | Expected | Actual |
|---|---|---|---|
| Register new user | {“email”: user@gmail.com, “password”: userPassword} | Status 201, {“message”: "Account successfully created!”} | As expected |
| Register existing user | {“email”: user@gmail.com, “password”: userPassword} | Status 400, {“message”: "Error - User already exists”} | As expected |
| Register user without email | {“email”: “”, “password”: “userPassword”} | Status 400, {“message”: "Error - you must supply both email and password"} | As expected |
| Register user without password | {“email”: “user@gmail.com”, “password”: “”} | Status 400, {“message”: "Error - you must supply both email and password"} | As expected |
| Login existing user | {“email”: user@gmail.com, “password”: userPassword} | Status 200, {“token”: JWT Token, “access-token”: JWT Token, “expires_in”: 86400, “token_type”: “Bearer”, “message”: “Successfully logged in!”} | As expected |

| Test Case | Input | Expected | Actual |
|---|---|---|---|
| Login existing user wrong password | {“email”: user@gmail.com, “password”: wrongPassword} | Status 401, {“message”: “User credentials do not match any of our records”} | As expected |
| Login non-existing user | {“email”: userNotExist@gmail.com, “password”: userPassword} | Status 401, {“message”: “User credentials do not match any of our records”} | As expected |
| Login user without email | {“email”: “”, “password”: “userPassword”} | Status 401, {“message”: "Error - you must supply both email and password"} | As expected |
| Login user without password | {“email”: “user@gmail.com”, “password”: “”} | Status 401, {“message”: "Error - you must supply both email and password"} | As expected |
| Offences helper | n/a | Status 200, {“offences”: ["Advertising Prostitution”, "Armed Robbery”, …]} | As expected |
| Areas helper | n/a | Status 200, {“areas”: [ {List of areas} ]} | As expected |
| Ages helper | n/a | Status 200, {“ages”: [ {List of ages} ]} | As expected |
| Genders helper | n/a | Status 200, {“genders”: [ {List of genders} ]} | As expected |
| Years helper | n/a | Status 200, {“years”: [ {List of years} ]} | As expected |
| Search without authorisation header | offence=Arson | Status 401, {“message”:“Authorization token not supplied in header”} | As expected |
| Search without offence parameter | Authorisation: {Valid token} | Status 400, {“message”:“You must present a valid offence to search”} | As expected |
| Search with correct offence parameter | Authorisation: {Valid token}, offence=Arson | Status 200, {“query”: {“offence”: “Arson”}, “result”: [ { Result data } ]} | As expected |
| Search with correct offence parameter and incorrect filter parameter | Authorisation: {Valid token}, offence=Arson, age=chile | Status 400, {"Message": "Error present in parameters”} | As expected |
| Search with correct offence parameter and filter | Authorisation: {Valid token}, offence=Arson, age=Adult | Status 200, {“query”: {“offence”: “Arson”, “age”: “Adult”}, “result”:[ { Result data } ]} | As expected |

These tests and messages are based on what I have previously been expecting my server to respond with based on the inputs. It has since now changed to mirror those that can be found on the hackhouse server. When presented with an invalid filter parameter my server will respond with status 500 and message of an issue in parameters, whereas hackhouse would respond with 200 and return an empty result. This has been done as an improvement to provide information to developers with this API to know there is an error in them passing a filter that doesn't exist in the database.

## Appendix

Installation guide (based on MacOS):

- Install MySQL and import the provided database.
- Change directory to the root location of the Api files eg.

```
cd /Documents/api
```

- Run the following command, this will install all the necessary package dependencies

```
npm install
```

- Now run the `ifconfig` command and take note of the IP address of your computer, this is important for the next step
- Then run the following command, this will generate a self-signed certificate to allow for https requests.

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

  - Enter the information in the following the prompts. In the common name field enter the IP address of your computer from earlier. You will now have a certificate and key for https communication.

- Go into the knexfile.js is using the correct password for your MySQL root user
- Now you can run `npm start` and the API will now be accessible via your IP address that you took note of earlier