2020

CAB230 Stocks API – Server Side

Dylan Buckley

N10234977

6/10/2020

Contents

Introduction

Purpose & description

The stocks API is an express.js based server which was developed to provide an easy way to request certain stock information. It is designed as a RESTful web service which allows multiple different platforms to get information. Due to this, all responses from the API come in a JSON format. Using JSON allows the transfer of data to be painless while providing better readability for the end user, unlike alternatives such as XML.

The API provides a range of different endpoints to allow consumers to get the data they need. Some of these end points require the user to be authenticated, which makes the API secure from unauthenticated requests. This authentication system has been thought out thoroughly to avoid complexity with making requests. This complexity can arise when verifying authenticated users on the API's server side. Our solution to this complexity is to use JWT tokens for one login that gets sent in the header of the request, for each request. That way if the token has not expired, you will be able to authenticate your request from any platform (browser, device etc.) at any time. This avoid issues with the usual sessions & cookies where only the application which stores them can be authenticated. Using this authentication methodology, we find that the process of authenticating requests for the API is very intuitive and simple for users.

In summary, the stocks API is a well-developed RESTful web service that provides users with relevant stock information. It can be used by both authenticated, and unauthenticated users but the latter is restricted to use only certain endpoints. This authentication system is handled by JWT tokens in the header of the requests, which provides a more flexible way for cross platform devices to request the data. Once requests are made, the express.js based API responds with JSON data for the user to then use how they wish. To learn more information about how the API requests work, Swagger documentation is provided on the homepage of the API.

Completeness and Limitations

All endpoints for the stocks API are fully functional. The stock API tester was used against the API and was found to have passed all 93 tests with 0 failures or pending requests. Below is a list of all the routes that are fully functional which have been implemented:

/stocks/symbols

/stocks/symbols?industry={industry}

/stocks/{symbol}

/stocks/authed/{symbol}

/stocks/authed/{symbol}?from={date}&to={date}

/user/register

/user/login

All optional queries work with the correct corresponding endpoint. All endpoints that also require JWT authentication are fully functional. All error responses are also properly handled for each endpoint. From the stock API testers results & manual testing of each endpoint, it is believed that there are currently no limitations with all the endpoints.

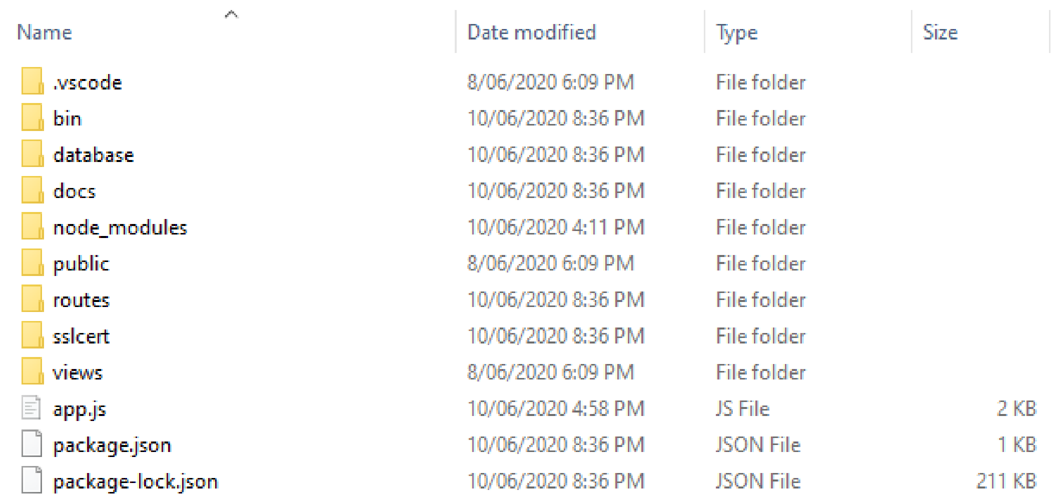
As for the rest of the assignment specification completeness, it is believed that the MySQL database, Swagger docs, Knex, helmet, morgan and HTTPS implementations for the API all are fully functional. Therefore, the claim is that all implementations required of the assignment specification have been met.

Modules for API

No additional modules used

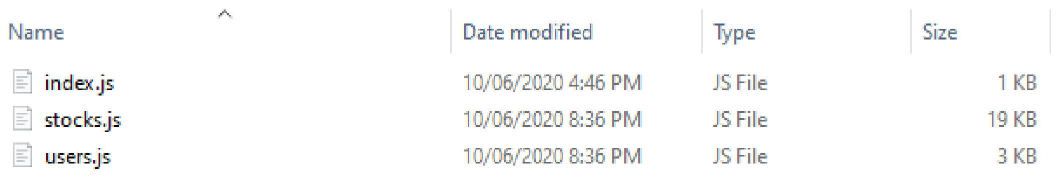
Technical Description

Architecture



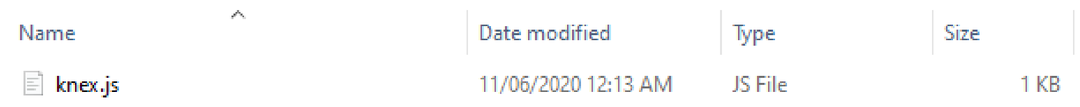
Name	Date modified	Type	Size
.vscode	8/06/2020 6:09 PM	File folder	
bin	10/06/2020 8:36 PM	File folder	
database	10/06/2020 8:36 PM	File folder	
docs	10/06/2020 8:36 PM	File folder	
node_modules	10/06/2020 4:11 PM	File folder	
public	8/06/2020 6:09 PM	File folder	
routes	10/06/2020 8:36 PM	File folder	
sslcert	10/06/2020 8:36 PM	File folder	
views	8/06/2020 6:09 PM	File folder	
app.js	10/06/2020 4:58 PM	JS File	2 KB
package.json	10/06/2020 8:36 PM	JSON File	1 KB
package-lock.json	10/06/2020 8:36 PM	JSON File	211 KB

The architecture of the application has been thought out to try and provide easy navigation and to follow good coding practices. Firstly, routing is done within app.js and is split up into multiple different router files depending on the use case. As seen in the image below, each route has its own JS file within the routes folder:



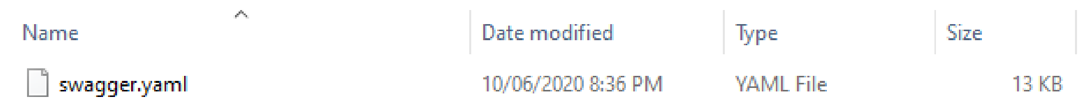
Name	Date modified	Type	Size
index.js	10/06/2020 4:46 PM	JS File	1 KB
stocks.js	10/06/2020 8:36 PM	JS File	19 KB
users.js	10/06/2020 8:36 PM	JS File	3 KB

Following this organization, the database connection file is also within its own folder called database:



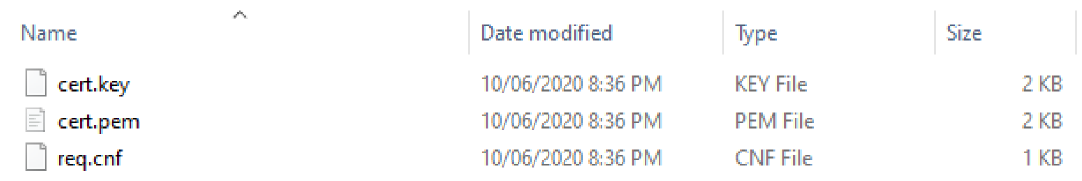
Name	Date modified	Type	Size
knex.js	11/06/2020 12:13 AM	JS File	1 KB

Underneath the database folder is the docs folder, which is used for handling the swagger documentation shown in the index.js route:



Name	Date modified	Type	Size
swagger.yaml	10/06/2020 8:36 PM	YAML File	13 KB

Lastly, the sslcert folder is used to store the TLS certificate information that has been generated by the server. The folder includes the config file, the certificate and the private key:



Name	Date modified	Type	Size
cert.key	10/06/2020 8:36 PM	KEY File	2 KB
cert.pem	10/06/2020 8:36 PM	PEM File	2 KB
req.cnf	10/06/2020 8:36 PM	CNF File	1 KB

Security

Security is a major factor when developing a RESTful API. Therefore the following security measures have been implement to prevent basic mishandling of the API:

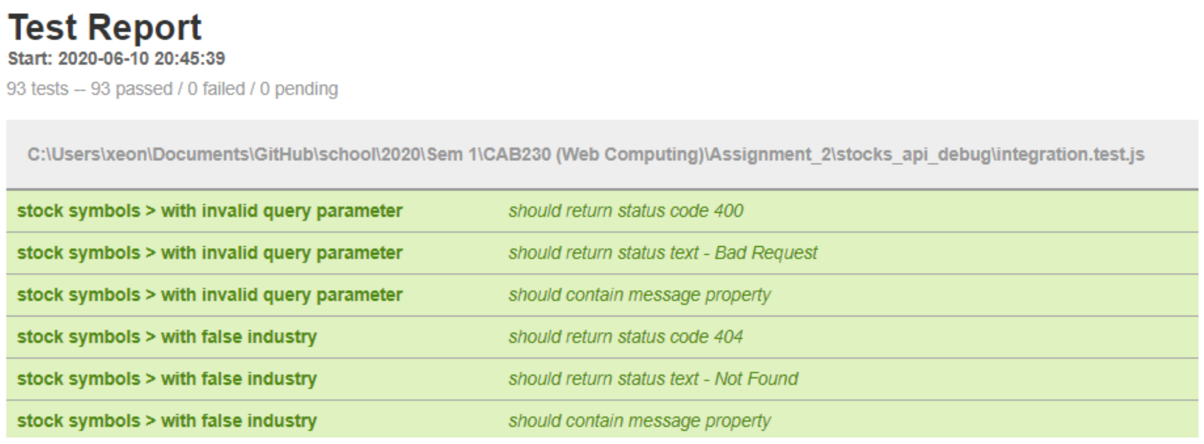
- Knex query builder to avoid raw SQL (SQL Injections)
- Helmet with default settings enabled:

  (Helmet's default settings list is found here: https://github.com/helmetjs/helmet)

- Morgan with a logging level of 'common'
- Bcrypt used for hashing user passwords
- JWT tokens for authenticating requests in the header

Testing

The stocks API tester was used to test each endpoint of the stocks API, this is the html report output:



Test Report
Start: 2020-06-10 20:45:39
93 tests -- 93 passed / 0 failed / 0 pending

C:\Users\xeon\Documents\GitHub\school\2020\Sem 1\CAB230 (Web Computing)\Assignment_2\stocks_api_debug\integration.test.js

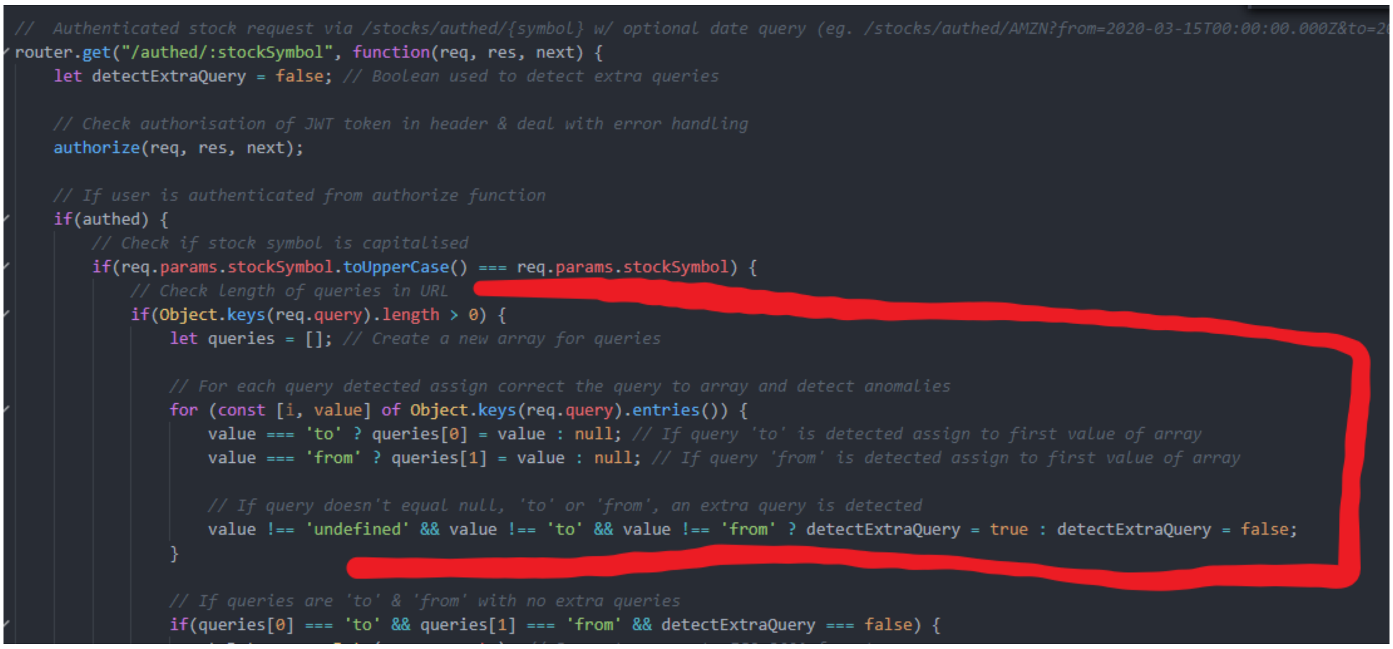
stock symbols > with invalid query parameter	*should return status code 400*
stock symbols > with invalid query parameter	*should return status text - Bad Request*
stock symbols > with invalid query parameter	*should contain message property*
stock symbols > with false industry	*should return status code 404*
stock symbols > with false industry	*should return status text - Not Found*
stock symbols > with false industry	*should contain message property*

Difficulties / Exclusions / unresolved & persistent errors /

All functionality for the API has been completed to meet the assignment specification. Therefore, there are no functionalities which have not been finished that are required. As far as I have tested also, there are not any bugs in the API, but I am sure there are somewhere! As for the technical roadblocks while implementing these functions, there are many.

Firstly, I completely forgot that the one of the practical worksheets outlined how to extract query results from the URL queries. Due to this mistake, I implemented my own custom algorithm for determining query results and anomalies. This algorithm perfectly matches the test API, even down to the swapping of queries (eg. to={date}&from={date} === from={date}&to={date}). It also matches other little quirks about the test API's URL handling to try and exactly match specification. It will also will give a Boolean "detectExtraQuery" for any queries that are not desired and then can be dealt with in error handling. This algorithm however complexifies and uses a lot more code than the practical worksheet method for extracting queries. However, I only realized this after the fact, and due to my algorithm working perfectly for the assignment specification, it has been left the same.

Below you can see how unnecessarily long the algorithm (highlighted in red) is for determining queries in my API:



```javascript
//  Authenticated stock request via /stocks/authed/{symbol} w/ optional date query (eg. /stocks/authed/AMZN?from=2020-03-15T00:00:00.000Z&to=2
router.get("/authed/:stockSymbol", function(req, res, next) {
    let detectExtraQuery = false; // Boolean used to detect extra queries

    // Check authorisation of JWT token in header & deal with error handling
    authorize(req, res, next);

    // If user is authenticated from authorize function
    if(authed) {
        // Check if stock symbol is capitalised
        if(req.params.stockSymbol.toUpperCase() === req.params.stockSymbol) {
            // Check length of queries in URL
            if(Object.keys(req.query).length > 0) {
                let queries = []; // Create a new array for queries

                // For each query detected assign correct the query to array and detect anomalies
                for (const [i, value] of Object.keys(req.query).entries()) {
                    value === 'to' ? queries[0] = value : null; // If query 'to' is detected assign to first value of array
                    value === 'from' ? queries[1] = value : null; // If query 'from' is detected assign to first value of array

                    // If query doesn't equal null, 'to' or 'from', an extra query is detected
                    value !== 'undefined' && value !== 'to' && value !== 'from' ? detectExtraQuery = true : detectExtraQuery = false;
                }

                // If queries are 'to' & 'from' with no extra queries
                if(queries[0] === 'to' && queries[1] === 'from' && detectExtraQuery === false) {
```

Installation guide

Once the source code has been downloaded, make sure you are in the top-level directory for the server. This is the same directory level where the app.js file is located. Then execute the required commands below to get the server up and running:

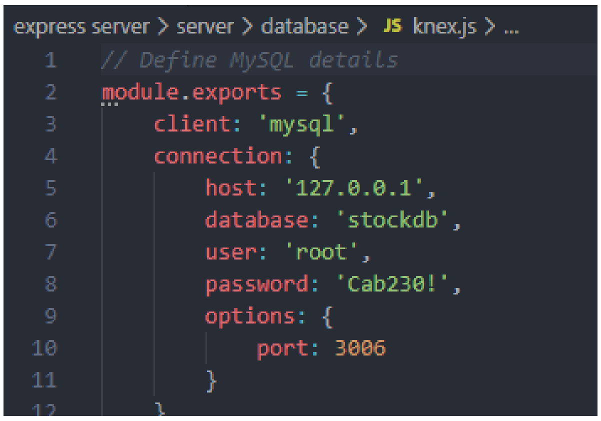
1. Install node modules used by the server:

```
PS C:\Users\xeon\Documents\Assignment_2\express server\server> npm install --save
```

2. Once the node modules are installed, the server can be started:

```
PS C:\Users\xeon\Documents\Assignment_2\express server\server> npm start
```

3. At this point, you should be able to navigate the https://localhost to view the running server. However the database is not setup & all mysql queries will result in an "Error executing MySQL query" response. To fix this, the database must be imported to MySQL (my stocks.sql file includes the users table). Once it is imported, the settings in the knex.js file (database connection file) must be changed to match your local setup. An example of the knex.js config for the QUT Linux VM's can be seen below:



```javascript
// Define MySQL details
module.exports = {
    client: 'mysql',
    connection: {
        host: '127.0.0.1',
        database: 'stockdb',
        user: 'root',
        password: 'Cab230!',
        options: {
            port: 3006
        }
```

References

No references or resources were used to write the introduction or any other part of this document.