**Database**

* <https://www.verywellmind.com/what-is-a-mood-tracker-5119337>
* Entity Discovery done on daylio, moodfit, Mood Tracker apps
* Initial rough ERD

Figure this doesn’t account for multiple users

Graphical user interface

Description automatically generated

* Database Naming conventions lowercase/ snake-case
  + **Tables**: tbl\_mood, tbl\_entry etc
  + **Attributes**: no qualifier
    - tbl\_mood.name, not tbl\_mood.mood\_name
  + **PKs**: table\_id
    - tbl\_entry.entry\_id
  + **FKs**: optional descriptor name of pk
    - tbl\_activity.icon\_image\_id = tbl\_image.image\_id
* Muliple activities can be chosen for an entry

**Database API**

Worked on routing. Got a basic sort of switch based routing going but not happy about it.

**Web Server**

Inside httpd.conf I set the following:

<Directory />

FallbackResource /uni/project/index.php

</Directory>

To ensure that the index.php page can be used for routing as per normal modern web routing standards.

Renamed index.php to http-controller.php

Having trouble getting the root of the webserver, eg localhost/uni/project/

**Login system**

Decided to store login information in a JWT. The user provides a password, then the system pulls the users password out of db, extracts salt, and reencrypts the password entered in login form, then compared the two encrypted passwords for equality. If equal the system creates a JSON web token with a payload that contains the user’s id and username from the database. The payload also contains an expiry property, which is a unix timestamp that is 1 hour after the creation time of the token. This means tokens are automatically invalid after an hour. When the system encounters one of these expired tokens during authentication on the server side, it deletes the token, effectively logging out the user.

**Choosing Tech Stack**

First looked at php only, then at php-express (express written in php), then moved to a fully express based routing.

MariaDB/ MySQL

ExpressJS

NodeJS

**Libraries / imports / frameworks**

* Express – a nodejs module for server routing
  + Ejs – templating/ view engine for express
  + Express-session – a nodejs module for using sessions in express
  + Body-parser – a nodejs module for parsing HTTP bodies (eg json, form encoded) in express
  + Mysql – for communicating with a mysql server
  + Babel – to allow jest to work with typescript

**Process**

**Start – 27/11/22**

**Stack**

First few weeks were spent playing with various stacks, initially I had tried a php only route, but it felt dated, so I tried php-express framework, which is a version of express in php language. This worked well but it also felt dated. Finally I landed on using NodeJS/ Express JS as the backend framework.

The stack at this point is as followins, MySQL, Express, Vanilla JS and Node JS (budget MEVN :D)

**27/11/22**

**Testing**

Unit testing with Jest, and functional testing with Supertest/ Jest. Initially I did some design and development without any thought to unit or functional testing, but as the app grew it became apparent that managing complexity would be a problem. I stepped back and researched some unit testing and functional testing strategies for Express Apps and decided that I would use jest (for unit tests and functional tests) and supertest (for functional tests).

**29/11/22**

**TypeScript**

Moved to typescript for backend code. I’m more comfortable with a strong typing system for development to minimiise the “squishy” nature of js

# Error Handling

1. Database connection error handling
   1. When db not active when started.
   2. When db not active suddenly when running.

# Authentication

## Login

|  |
| --- |
| (browser -> web app)  POST /login HTTP/1.1  Host: <http://localhost:3000>  Content-Type: application/x-www-form-urlencoded  username=exampleusername&password=examplepassword |
| (web app -> API)  POST /api/auth/login HTTP/1.1  Host: <http://localhost:3000>  Authorization: Bearer *<<API key from environment variabless>>*  Content-Type: application/x-www-form-urlencoded  username=exampleusername&password=examplepassword |
| Figure Example POST requests for login routes |

The web application manages authentication using JSON Web Tokens (JWTs). The user navigates to the login form, enters their username and password, and submits. The form then sends a HTTP POST request to the /login route of the web app. Internally this route‘s middleware packages up the submitted username and password into a new POST request to be sent to the API’s /api/auth/login route.

This second POST request is authorized using the Authorization header. A unique secret API key is read from environment variables (or in my case a dotenv library .env file that contains key value pairs to be injected into the process.env global node object as if they were environment variables), and added to the Authorization header (figure x).

The API receives this request and attempts to authorize the request via the middleware authenticateRequestBySource.

This middleware reads the Authorization header from the request, then queries the database tbl\_key table to check if the key valid – i.e. that is it both exists and has its active bit(1) attribute set to 1.

* If the key is valid (present and active), the request is authorized by the middleware and next() is called to move onto the next route specific middleware.
* If the key is not valid, then the API returns a status code 401 (Bad Request) and an accompanying JSON response body (modelled by my SuccessResponse class) indicating the request is not authorized and was rejected. This logic will be discussed below.

Once the middleware has authorized the request the login middleware is called. This middleware is mounted in the /api/auth/login POST route in the API’s authRouter.

This middleware then processes the request as follows:

The POST body is validated to contain the required properties (username and password). If it does not, a 401 response is returned with a JSON body modelled by my LoginResponse class, which indicates the login attempt’s success with a boolean, and optionally token and error properties. The latter being an array containing various validation error messages (eg. “no username provided”).

If the post body is valid, then the middleware calls a AuthApiDataAccessObject (auth DAO)– also named login, passing in the provided username and password as parameters. This data access object exists to maintain separation of concerns. All authentication database queries/ statements are housed in this data access object.

This auth DAO first confirms the user exists by querying the database for user data for users with the username equal to the POST body username property.

If the user does not exist, a 401 is returned with an appropriate error message (i.e., “This username does not exist”). If the user does exist then the password submitted during log in is hashed with the same salt as the stored password from the database, and the submitted and stored passwords are then compared for equality.

If the username and password combination is accepted, the API constructs a JWT with a payload containing the user’s username, email, and id (id primary key from DB), and an expiry date/time stamp in the form of a Unix time stamp.

|  |
| --- |
| {      "success": true,      "token": "eyJhbGc…0Bi31Cg"  } |

Figure Example Success Response (JWT shortened for readability)

The API returns a status of 200 with a JSON body to the web app that contains the JWT as one of its properties (e.g., figure x). The web app receives this response, and if the success property is true, retrieves the token from the response body, and stores in a cookie on the user’s machine named token.

## Restricted Areas

When a user attempts to access a login-restricted part of the web app, a middleware function named authenticate then extracts the JWT from the token cookie and attempts to verify it. It does so by attempting to decode the token into a plain JSON object using a secret key.

|  |
| --- |
| function verifyToken(token: string): JwtPayload {    return jwt.verify(      token,      process.env.MOODR\_TOKEN\_SECRET!    ) as JwtPayload;  } |
| Figure the jsonwebtoken node module allows us access to the verify method. This method allows us to decrypt and read the payload of the JWT using a secret. In my project you can see the secret (2nd arg – in red) is read from environment variables/ dotenv .env file |

In my project this secret is in the form of a single unchanging value that is accessible in my jwtHelpers.ts module (figure x). This module is imported by both the web app and the API, allowing both to process these JWTs in the same manner. A better system would be to use a system of changing/ rotating keys that can be enabled/ disabled as needed. This would likely take the form of another database table, with the keys and their permissions stored in attributes.

## API Authentication

The API has a primary method of authorization for requests originating from the server/ back-end and another secondary method for HTTP requests that originate from the web applications client-side scripts.

The primary method of authorization is via that of an API key set in the Authorization header. This method was discussed in the Login section above.

The secondary method of authorization is used currently by only one file (namely public/script/chart.js). Initially the /visual route that served the template that used this chart.js script used the primary method discussed above. This caused the web application to do a series of large, unwieldy and sometimes unnecessary database queries (i.e., separate database queries for mood frequency data, mood valence data, mood arousal data, and activity/ mood relationships all within a single controller method). This was done so that the data could be passed to the templates safely in my controller methods on the server without exposing any API endpoints, or API keys to the client side scripts.

I decided that it would be much preferrable to call a single charts data at a time in an ajax-like fashion from within the client side script that controlled the visualisation charts. This means that only the data specifically required is queried from the database.

The visualizeRouter utilised by the API uses a different authentication middleware/ methodology to the rest of the API, namely the authenticateRequestByJwt.ts module.

Instead of authorizing requests via API keys, the users login token cookie is sent in the Authorization header. As discussed briefly in the login section above, this token is verified via my jwtHelpers.ts module (which wraps the jsonwebtoken npm module’s verify and sign methods). If not verifiable as a valid JWT a 401 Not Authorized is returned. If verifiable, the user’s id (their primary key in the database) is read from the decoded JWT’s payload and stored in an ExpressJS local response variable named res.locals.userId, and next() is called which then passes control to the route specific middleware (aka controller methods), where this local response variable is accessible.

If this authentication middleware authorizes a request, one of the visualizeRouter’s several chart data retrieval routes is then executed, returning that subset of the data to the client-side script, which processes the data and renders the chart.

wimplemented so that the visualization section of my web application need not make one large and unnecessary API acll

that is a in the Authorization header as a Bearer token. DB API KEY IMPROVEMENT CRAIC HERE. where it can access the username, email, id and expiry data within. This API key is accessed as an environment variable on my PC, which enables me to keep it from appearing within the repository, a relatively common security lapse.

This key is kept in a .env file in the root directory of the project. It is then injected into the process.env global object in node, where environment variables are read from the system and stored for use in node applications.

This API key is available only to my web app’s backend, meaning the only resource authorized to consume my API is the web app, or any other application I share this key with. In this way the security of the database, and the data within is maintained.

If the current time exceeds the expiry time of the JWT then the token is cleared from the user’s computer, logging them out.

|  |
| --- |
| const opts = {          method: httpMethod,          body,          headers: {              'Content-Type': 'application/x-www-form-urlencoded',              'Authorization': 'Bearer ' + process.env.REQUESTOR,              ...(token && { 'Cookie': 'token=' + token })          }      };  const fetchResponse = await fetch(buildApiUrl(endpoint), opts); |
| Figure How the web app makes requests to the API – note the process.env.REQUESTOR environment variable which contains the API key. |

In this snippet, takEN from my apiCall() method utilized by the website backend to make HTTP requests to my API, we can see highlighted in red, where this API key is read from environment variables and used as a Bearer token for the API.