Peter Marley – Student number 13404067 – email pmarley03@qub.ac.uk

Video report: https://www.youtube.com/watch?v=h2KZUa5ZVyU

Moods, Triggers and Visualisation

I used a modified version [1] of Russell's Circumplex Model of Affect [2] as the mood model for my project. I considered several other models.

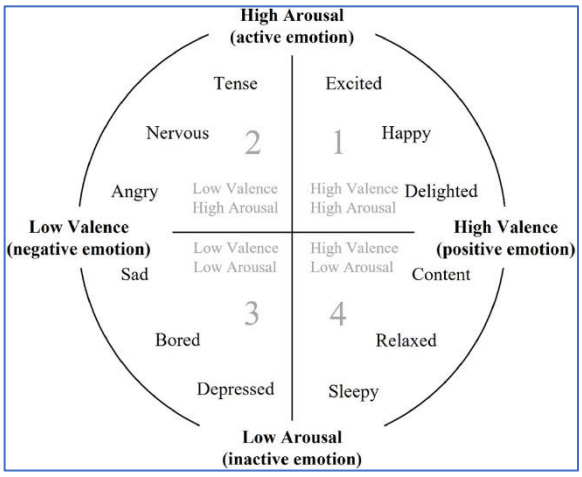


Figure 1 Russell's Circumplex Model of Affect (Modified)

The Valence-Arousal-Dominance model [2] (also proposed by Russell) was a front runner, but it's third dimension (dominance) added an unnecessary complexity that I felt would not add enough value to the project to warrant handling this complexity.

Another strong contender was the Ekman's Basic Emotions Model [3] which attempts to enumerate the discrete emotional states categories, i.e. anger, disgust, fear, happiness, sadness, and surprise. This would fit well within the constraints of this project, but it seemed reductive from a user's perspective.

Russell's Circumplex Model of Affect was particularly attractive as it allowed 2 axes (valence and arousal) with which to classify each "mood" - see Figure 1. This would be useful during visualization of mood logs, as it attaches data potentially useful to the user to each emotion. I used a modified version of this model which classified 12 separate moods, 2 valences and 2 arousal levels.

For mood triggers/ context I used *activities* the user engaged in (i.e., those that prompted the mood) and a free-text *notes* field. These activity triggers are sub divided into groups. Each user account, upon creation, is seeded with an Activity Group named *Default*, in which are nested two Activities – *Work* & *Exercise*. User's may add and remove activities themselves via the main menu to customize their experience. This will be demonstrated in the video report.

This allows the user's to associate their day to day activities as triggers for certain moods. As the user creates more and more logs, my visualization charts (Activity/ Mood relationship chart especially) will allow users to spot these trends over time.

To visualise the moods, I used the Chart.js library [4], which is included via a script tag's *src* attribute in *visual.ejs*. I created a set of uniquely authorized routes on my API (in the *visualRouter.ts*). These routes are authorized not by API keys like the other API routes, but instead via the user's login token JSON Web Token cookie (JWTs - discussed later) set in the *Authorization* header of the request. This was done so that the visualization client-side JavaScript scripts (in *./app/public/script/chart.js* – not to be confused with the chart.js library) can call each chart's endpoint as and when it requires the data (in an ajax-like fashion), instead of doing a potentially wasteful large query to gather all charts data in one go. I had to authorize these requests originating on the client-side without exposing API keys so I chose this method.

This approach ensures that logged-in users can only access their own chart data as the user ID used in SQL queries is derived from the JWT token's encrypted payload's *id* property.

On all charts, Positive/ Negative mood valence is denoted with a green or red colouring respectively to make it obvious immediately which days were broadly positive, and which were negative.

For the actual statistics rendered in the charts I decided on the following:

(1) Raw mood frequency bar chart. The frequency of each mood in that user's log history is shown. A quick glance can show the user what their most common moods are.

(2 + 3) arousal and valence pie charts. These give an indication of how common each valence / arousal [2] in mood logs. This gives the user an immediate accounting of whether their moods are mostly positive or negative, and weather they are of low or high arousal.

(4) Summary line graph. This that displays the most common mood logged on each day that has at least a single log entry. This shows the most common daily mood over the entire time span the user has been logging their moods.

(5) Activity/ mood relationships. That is a set of charts that relate the contextual *activities* the user recorded with the *mood* logged. When the main relationship tab is selected the user is shown a chart that shows the most common mood/activity combinations. This helps the user pinpoint what activities may be triggering specific moods regularly. This tab nests several sub-charts equal to the number of activities that user has on their account. Newly registered users will have only two sub-charts (for default Work and Exercise activities), but my *testuser1* demo account has eight charts under this tab, as there is currently 8 activities associated with that account.

There is also an Introduction tab, that briefly explains the concept of Russells' Circumplex model, links to the original paper, and gives a plain English description of what that means for the user.

Functional Requirements

User Registration: User registration is done via a registration form that has both client and server side validation. The form is clean and easy to read, and the client side validation uses input border/ background colouring and validation messages in absolutely positioned *div* elements, which are hidden via my *hidden* custom CSS class when the DOM is rendered, and this CSS class is removed via client-side JavaScript event handlers to show the validation error message. The form requires the user to confirm their password to ensure it is what they expect, and allows the user to click a checkbox to show the password if they wish. If a logged in user navigates to the register page (at */user/register*) accidentally, e.g., via the browsers back button, the registration form is not displayed, as to not confuse the user.

If user registration is successful, the second route controller middleware method (*attemptLogin*) is called which seamlessly logs that user in.

Upon reflection, it would have been better if I had included a *confirm email* field, and required the user to verify their submitted email address is under their control.

Logging In: As discussed in the video, a user is considered successfully logged in if their client has a cookie named *token* containing a JWT signed/ encrypted with the *MOODR_TOKEN_SECRET* encryption key read from environment variables. Even if a user attempts to create a fraudulent login cookie, they cannot sign the JWT with the same encryption key available server-side, making their fraudulent JWT invalid as the authorize middleware will fail to verify (decrypt and decode) the JWT. Thus only the web application/ API can create/sign and verify/decrypt these JWTs tokens.

I chose to render the login form via a modal, accessible via the hamburger button. This means the user can access the login form no matter what page they are on. If the user is logged in, then this modal instead shows the username and avatar of the user, with options to navigate to the account details page or to log out. The navigation menu is also displayed in this modal when the user is logged in, so that if the user is using the website via a device with a smaller viewport, and the CSS breakpoints hide the navigation menu on the main body of the page, they can still access it via the hamburger [5] button.

Mood Logging: The user can access the mood logging form via the web applications */mood/new* or */mood/<entryId>* GET routes. The form is dual purpose and depending on which of these two routes the form is accessed via, will allow the user to create a new log entry, or edit an existing log entry. The form allows the user to select from a series of positive and negative moods, as defined by Russell's Circumplex Model of Affect [2]. This makes the appropriate mood easier to find and makes the UI more intuitive for a user. Optionally, the user may leave some free-text notes and select the activity they were undertaking that triggered the mood. These *notes* and *activities* are discussed in the *Moods and Triggers* section above.

Mood List: The moods list page is accessible via the */mood/list* route. This page renders all the user's mood logged. The data is processed on the server and sorted by day. Each day's entries are then sorted chronologically to make navigating the list intuitive for the user. Here the user can easily *Edit* and *Delete* each entry by selecting the relevant button on that entry.

Edit Mood: When a user selects the *Edit* button on the mood list page, the add/edit entry form (at *./app/views/components/mood-entry-form.ejs*, embedded into the *./app/views/mood-entry-edit.ejs* template) is

displayed. This form will not allow a user to change the mood value or timestamp of the entry as per specification, but it does allow the user to edit the *notes* and *activities* mood triggers/ context data as they wish.

Delete Mood: When a user selects the Delete button on the mood list page, the web application send a DELETE request to the */api/mood/<userId>/<entryId>* route, and simply renders the mood list page once complete (reusing the controllers *getEntryList* method, which is used to get the data for the mood list page).

Mood Summary: The user can navigate to the */mood/visual* route, to access mood summary and visualisation charts. They are initially greeted with the abstract from Russell's paper [2], with a link to the paper online and a very small paragraph that attempts to explain the relevance of this model in plain English. The user can then click buttons along the top that will render the individual charts.

The *Activity/ Mood relationships* button renders a chart that shows all the most common combinations of activity and mood, and renders a series of buttons that allows the user to show the most common moods logged against each activity associated with that user's account.

The data from the charts, as already shown in the video, is requested directly from the API via client side JavaScript scripts, via a unique router that is authorized via the users JWT `token` cookie. This unique authorization method was employed so that the server need not request all visualization data in one giant query but may make smaller requests are needed for a specific charts data. Although this is an educational project, this was done as a nod to the needs of scaling such a web site up to potentially many users who have a great number of mood logs recorded.

Delete Account: A logged in user may click the hamburger button, and then the Accounts button on the displayed modal to access their account details page. I have not demonstrated username/ email changes as they are excess to requirements, although I did demonstrate the password change route as a successfully implemented PATCH request.

The account deletion process was demonstrated already in the video report. The reason I chose to require the user to type in their username to confirm account deletion was to make deletion itself a deliberate act, that cannot be executed by an errant misclick. I cannot imagine a scenario that would destroy a user's motivation to log their mood more than accidentally deleting their account and ALL data on it with a single accidental click.

Logout: A logged in user can click the hamburger button then the Logout button displayed on the modal. This route simply deletes the token cookie off the user's client (and sets the *authed* ExpressJS locals variable to false as a safety precaution as this ExpressJS locals variable is used by the web applications *authorize* middleware.

I placed this logout button the hamburger buttons menu modal, as it seemed prudent to allow a used quick and easy access to the logout functionality, no matter where on the website they were.

Authorization & Security

There are currently 3 main methods of authorization used in my project.

(1) Web application: The user's session is authorized on login restricted routes via the *authorize* and *restrictedArea* middleware functions from *./app/middleware/*. *Authorize* attempts to retrieve the user's login JWT *token* cookie from the browser, and then attempts to decrypt it using the *MOODR_TOKEN_SECRET* from environment variables. If the JWT is not valid (current time surpasses expiration time, or it was signed with a different secret), the *authed* ExpressJS locals variable is set to false. If it is valid, it is set to true. The *restrictedArea* middleware simple checks this *authed* locals variable is set to true for that request/ response cycle. If it is set to false the user is redirected to the */forbidden* route which displays a polite message informing the user they must be logged in. If *authed* is set to true, the *next()* ExpressJS method is called which calls the next middleware for that route. Note middleware functions can be mounted for the entire router/ application via *.use()*, or per route via the appropriate ExpressJS HTTP verb method (e.g., *.get*, *.post*) [5].

(2) API: All requests from the web application's back-end to the API are authorized via API keys. The web application reads the current in-use key from environment variables. The *dotenv* [7] node module/ library is used to inject key/value pairs (including the active API key amongst others) from the *./.env* file into environment variables. This *.env* file would not be committed to a repository as to not expose the API keys. The environment variables and their values in the real world would be added to the server machine when the application is deployed to a production environment, likely by CI/CD tooling. The keys are stored in the *tbl_key* table in the database and can easily be disabled by changing the *active* (*bit(1)*) attribute to 0 on a specific key.

These keys are passed in the Authorization header of the HTTP request. The API checks the key both exists in the database, and has its active bit set to 1, otherwise it rejects the key and does not authorize the request. This authorization method is enacted by the *./api/middleware/authorizeByRequestSource.ts* middleware.

(3) API: Requests from client-side scripts to the API's *visualizeRouter* GET routes are authorized via the user's login cookie named *token*. It is read from the client's machine and placed into the Authorization header of the request. The API reads this token, verifies & decrypts it, and the user's ID is read from the JWT's payload. The user id from the payload is then embedded into SQL queries. In this manner the client-side scripts can only successfully GET resources for an account they have the login details for.

I chose API keys to authorize requests from the web app to the API, as it is a well defined industry standard process, that has stood the test of time. My implementation allows immediately disabling of any keys which are compromised.

I chose JWT's to authenticate user's as they are also a well defined industry practice. It allows me to store information in the login cookie, ie, the user'd id, username, email and an expiry property. The expiry property is important as it means that a valid JWT has a limited lifespan, so even if a user has their JWT stolen off their machine, it will be invalidated automatically after 30 minutes.

This is a somewhat naïve implementation of handling expiry of tokens, as even if a user is logged in and operating the website, they will have to login every 30 minutes. I could solve this limitation in a number of ways, e.g. implementing a check that would automatically delete the cookie if the IP address using the token changed.

id	api_key	reason_for_use	active
1	bd12e269-5b53-4e0c-8807-99aecdfd9120	allows access to all api /user and /mood endpoints	1
3	2b0a2aec-0724-457b-8afd-b6b18c22efb6	demonstrating a disabled key	0

Figure 2 API keys stored

User Input Validation & Sanitization

As discussed already, user input is validated server side to ensure there are no characters in the user input that can be used to inject code into the website. This is accomplished via regular expressions that only allow safe characters past validation, and so if input passes the regex check it is therefore sanitized. These regular expressions are accessible throughout the server-side portion of the project via a configuration class named *Config* that reads data from JSON configuration files (Config and JSON files are viewable in *./common/config/*). The *Config* class stores the data in private fields and exposes it via getter methods, which stops the config data from being changed in the config object.

This Config class is imported in various files throughout the app/ API- and passed to client-side by passing the config data into ExpressJS locals variable. The config containing locals are then assigned to client-side JavaScript variables when rendering the template – viewable in *./app/views/partials/top.ejs* lines 14-21, and the middleware *./app/middleware/injectConfig.ts)*. This allows the client-side to access validation data for input fields such as *minLength*, *maxLength*, validation *regex* and a *description* containing a validation error message. Thus, to change all input validation throughout the app (or other config data such as the ExpressJS listening port), one must only change the data contained in the appropriate configuration file at *./common/config/*.json*. The validation regex is simple, it enforces a 8-20 character limit, and allowable characters of a-z A-Z and 0-9 on usernames; a 8-20 character limit, and allowable characters of a-z A-Z, 0-9, _, -, ?, ! and * on passwords; and emails simply have to be in the format example@website.topleveldomain (containing only a-z, A-Z and 0-9). In reality these regular expressions would need some further development to be production ready.

REST API Implementation

All database operations performed by the web application are done so through via RESTful API which, like the web application, is also an ExpressJS application. Technically my API is mounted as An ExpressJS Router in the main web application but it could easily be deployed as its own ExpressJS application. The API is divided into 3 ExpressJS Routers - *authRouter*, *moodRouter* and *visualizeRouter*, to maintain separation of concerns. These 3 routers are mounted in the *./api/api.ts* ExpressJS Router, which it itself mounted in *./app/app.ts* ExpressJS application. I already described Request Authorization in the Security section.

All database operations executed by the API are housed in my two database access objects (DAOs) in *./api/database/*. The *format* function, provided by the mysql2 node module/ library, allows us to prepare SQL statement with escaped parameters, to prevent SQL injection attacks. In concert with validation and sanitization of user inputs discussed above, this is an additional layer of protection against SQL injection and XSS attacks.

The RESTful API I developed for this project was in the form of an NodeJS/ExpressJS application wrapping a MySQL database running on an Apache Server, managed via XAMPP. For the purposes of a university project the API is mounted via the main web application's app.ts as an ExpressJS Router, although it could easily be deployed as a separate Express application.

This Router is defined in./api/api.ts. Its routes are divided into 3 groups, */user*, */mood* and */visual*, which are themselves ExpressJS Routers, each with its own area of concern. Authorization was discussed in its own section above, but the former two are authorized via API keys and the latter via a JWT.

Endpoints

All endpoints accept only *application/x-www-form-urlencoded* Content-Type in request bodies (via the *express.urlencoded()* middleware). All endpoints use the ExpressJS *Response* object method *.json()* [8] to convert from TypeScript/ JavaScript response objects to JSON (internally via the *JSON.stringify()* method). These response objects are stored in the *./common/response/* folder. Most of them extend *SuccessResponse*, barring some specific use case response objects such as *EntryDataResponse*. All values in red are numeric request parameter values that are changed per request to target specific resources (i.e.,specific user, specific log entry).

```
POST /api/user/login
POST /api/user/register
DELETE /api/user/userId
GET /api/user/userId
PATCH /api/user/userId
PATCH /api/user/userId/password
GET /api/mood/userId/new
POST /api/mood/userId/new
GET /api/mood/userId/list
GET /api/mood/userId/entryId
PUT /api/mood/userId/entryId
DELETE /api/mood/userId/entryId
GET /api/visualize/moodFrequency
GET /api/visualize/arousal
GET /api/visualize/valence
GET /api/visualize/relationship
GET /api/visualize/summary
```

Additionally, there is a fallback route that catches requests that could not be matched to a route and returns a 404 with an appropriate error message.

<ANY HTTP VERB/METHOD> /api/*

Database Design

I made significant modification to the provided database layout (see Figure 3). I tried to incorporate the lessons from the Database module about normalising data, reducing data redundancy, and attempting to reduce opportunity for data anomalies/ corruption.

The table *tbl_entry* is my implementation of the table *moods* from the design in the specification. I normalised out the *value* attribute from the specification into its own table (*tbl_mood*), as to introduce the 12 moods identified in my modified version of Russell's Circumplex Model of Affect. To expand the number of moods available for users, as some versions of Russell's model have significantly more than 12 moods, more can simply by inserted into *tbl_mood*, and so will be immediately available for use by all users. The *tbl_mood_arousal*, and *tbl_mood_valence* lookup tables exist only normalise out the valences and arousals out of *tbl_mood*, and contain only two records apiece. *tbl_mood_image* exists only to hold the URLs for the mood's icons (the emoticon-like images associated with each mood).

The *context* attribute from the provided design has been splint into 2 pieces of information: *notes*, which is analogous to the specification's *context* field, and *activities*. Activities have a many-to-many relationship with mood entries and so a linking table named *tbl_entry_activity* was created to allow this linking of *tbl_entry* and *tbl_activity*.

tbl_entry_images was implemented to allow a user to attach a number of images to a mood log entry. While I have demonstrated an entry that has images attached, I did not implement the functionality to allow a user to do this. Nevertheless, the basic functionality is in place, and our API returns these images if they are present in the database. With more time and foresight, I would have implemented this to allow a user to attach images themselves, as extra context for an entry.

tbl_user_image, in the same way, is the basis for functionality that allows a user to set a personal avatar image for their account. Currently when a user registers, a default silhouette avatar URL is assigned to their account. The avatar image *img* tag's *src* attribute is populated from the URL stored in tbl_user_image and can be

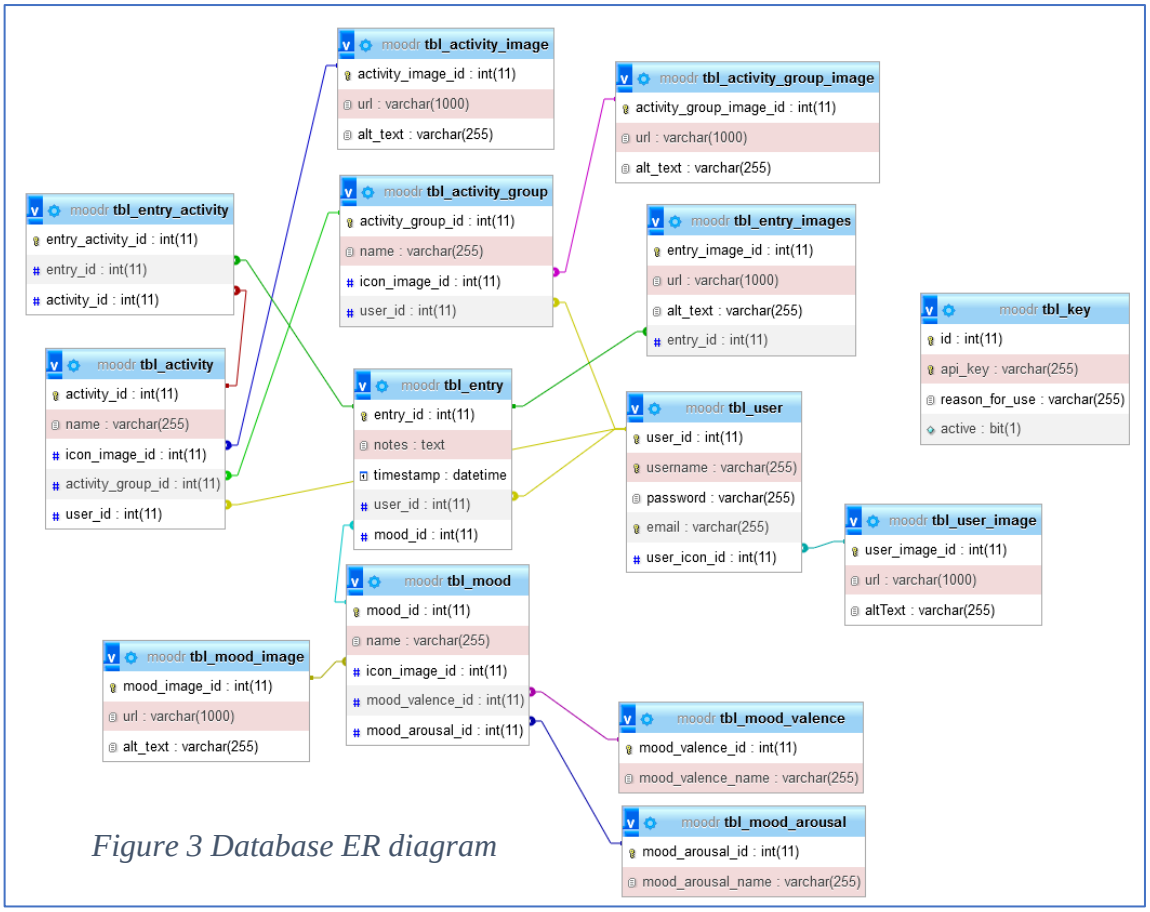


Figure 3 Database ER diagram

updated with an insert to tbl_user_image, and an update to *tbl_user*'s *user_icon_id* attribute. However, currently a user cannot access this functionality themselves.

All the image tables in the database have an *alt_text* attribute, which is set to the associate img HTML tag's *alt* attribute for accessibility.

The API storage table *tbl_key* has already been discussed, but it seemed appropriate not only to store the key itself, but its reason for creation, and the *bit(1)* active attribute was added to allow quick disabling of keys without deleting their associated record from the database.

Project Structure

In the video report and Appendix 2, you can see the file structure of my project, which is organized into four main folders: *api/*, *app/*, *common/*, and *ssl/*. The web application is in the *app/* folder, while the API is in the *api/* folder. Both folders contain ExpressJS applications that are separately configured for middleware, error handling, and routing in the *./app/app.ts* and *./api/api.ts* files. The *docs/* folder is where I stored my postman collection backups, database exports and my written report.

They both contain *controllers/*, *middleware/*, *routes/* and *utils/* folders. The routes folder contains the routers that define the actual routes for the app/ API. The *controllers/* folders contain the router controller methods, which the routers call sequentially in order of addition to process the request/response cycle. Router or app level middleware is stored in the middleware folder, as to highlight the difference between them and controller/ route level middleware specified in the controllers. The *api/utils/* and *app/utils* folders contain utility functions such as *apiCall* and *buildApiUrl* on the app side, and *crypt* (which provides cryptographic functions around hashing & salting passwords and checking submitted passwords during login) and *dbConnection* on the API side.

The *./app/utils/apiCall.ts* file provides an importable method via which my web application can make standardised calls to the API. The *dbConnection.ts* file which is imported to access the async function *getConnection*, which is a standardised way of accessing a MySQL connection pool to query the database. The two DAOs in *./api/database* use this function extensively.

The *./api/database* folder contains my DAOs as mentioned before, so that all direct SQL executing code can be isolated. The *./app/public* and *./app/views* folders store my static web content, and EJS templates respectively.

The *./common* folder contains classes (general, configuration and response objects), configuration data and general utilitity methods utilised by the web app and API, such as the important *jwtHelpers.ts* file which is imported and used for operations on JWTs.

The ./ssl folder contains the private key, certificate signing request and certificate used when enabling TLS/SSL encryption. *NODE_TLS_REJECT_UNAUTHORIZED* env variable is set to false so that I did not have to configure a root certificate authority, but the security of HTTPS/TLS was considered.

A *README.md* file has been included at the projects root folder, and explains how to run the application.

Frameworks & Libraries

cookie-parser [8] is a node module that allows ExpressJS to parse cookies attached to requests and embeds them in that request express.Request object where they can be used.

EJS [10](embedded Javascript templating) is a popular view engine for Express. It uses standard HTML syntax, with the ability to embed server-side javascript code that is executed when the template is rendered.

ExpressJS [11] is a lightweight web framework for node.js that has a well defined and mature routing system and numerous utility/ helper methods for managing the request/ response cycle.

node-fetch [12] is a fantastic module that allows an application running on node.js to access the Fetch Web API available to JavaScript that is running on a browser. This is the module I used to execute the actual http request from the web application to the API.

ts-node [12] allows me to execute TypeScript code without requiring a build step to compile TypeScript into JavaScript prior to execution. TypeScript is a superset of JavaScript that has extra features over vanilla JavaScript. The primary functionality of interest to me was strong typing and the ability to create interfaces which is not possible in vanilla JavaScript. I find the dynamically typed nature of JavaScript, and the problems that it can cause when passing around variables, to be a serious negative when considering JavaScript.

crypto-js [14] is a library that allows me access to strong cryptographic functions. I used this library to access the CryptoJs.SHA256 method to create strong professional grade one-way cryptographic hashes which I utilised to salt and hash passwords before storage in the database, and to salt and hash (with the stored passwords salt) passwords submitted during login so that they may be compared with the stored value.

jsonwebtoken [15] is a library that provides functionality for signing (encrypting) and verifying (decrypting) JSON Web Tokens, which my project stores in cookies on the users machine to allow authorization of logged in users.

method-override [16] is express middleware gives my project the ability to use forms (and links such as anchor element) for more than just GET or POST requests, by specifying a method override request parameter that tells the ExpressJS application to route that specific request to a specific HTTP methods route. E.g., an anchor element sets its href attribute to "/mood/145?_mo=DELETE" and if method-override has been added as application/ router level middleware, and correctly configurated to check GET request and to look for the "_mo" request parameter, Express will route to the DELETE method of that route instead of the standard GET request that an anchor elements are normally restricted to.

mysql2 [17] is a popular library that allows node.js to connect to and execute statements on a MySQL server. This library's methods are how I interact with my database from my DAO objects.

validator.js [17] is a string validation and sanitization library. I use this in a limited fashion, simple to use the validator.escape and .unescape functions for escaping HTML reserved characters from strings.

The only client-side framework/ library that I use is chart.js [4]. chart.js is a simple low-configuration charting library that renders chart data to HTML5 canvas tags, which is a performant and modern way of rendering simple graphics to a HTML page. I used this library to render my visualization charts.

dotenv [19] was used to inject values into environment variables from the .env file as discussed already.

The following @types [20] npm modules were added as npm dev dependencies to allow VSCode to implement *intellisense* and provide support for TypeScripts static typing system when developing: *@types/cookie-parser, @types/crypto-js, @types/express, @types/jsonwebtoken, @types/method-override, @types/mysql2, @types/node-fetch, @types/validator*.

nodemon [21] was installed as an npm dev dependency. This brilliant module automatically restarts the server when it detects changes in filetypes specified in the package.json "npm run start:dev" script, if the server was started with the *npm run start:dev* command.

Appendix 1 - References

[1] Unknown, “Modifier Russell's Circumplex Model of Affect,” [Online]. Available: https://ascelibrary.org/cms/asset/4f61d617-0684-42de-9120-1cb8d15e3734/figure1.jpg.

[2] “The circumplex model of affect: An integrative approach to affective neuroscience, cognitive development, and psychopathology,” [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2367156/. [Accessed 01 February 2023].

[3] P. Ekman, “Universals And Cultural Differences In Facial Expressions,” in *Nebraska Symposium on Motivation,* 1971.

[4] Chart.js, “Chart.js | Open source HTML5 Charts for your website,” Chart.js, [Online]. Available: https://www.chartjs.org/. [Accessed 15 02 2023].

[5] Wikipedia, “Hamburger button,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Hamburger_button. [Accessed 03 March 2023].

[6] ExpressJS, “Using Express middleware,” ExpressJS, [Online]. Available: https://expressjs.com/en/guide/using-middleware.html. [Accessed 22 March 2023].

[7] dotenv, “dotenv - npm,” [Online]. Available: https://www.npmjs.com/package/dotenv. [Accessed 3 January 2023].

[8] ExpressJS, “Express 4.x - API Reference - res.json(),” [Online]. Available: https://expressjs.com/en/4x/api.html#res.json. [Accessed 19 March 2023].

[9] d. dougwilson, “cookie-parser - npm,” [Online]. Available: https://www.npmjs.com/package/cookie-parser. [Accessed 22 March 2023].

[10] EJS, “EJS -- Embedded JavaScript templates,” EJS, [Online]. Available: https://ejs.co/. [Accessed 22 March 2023].

[11] ExpressJS, “Express - Node.js web application framework,” [Online]. Available: https://expressjs.com/. [Accessed 22 March 2023].

[12] Node-Fetch, “node-fetch - npm,” [Online]. Available: https://www.npmjs.com/package/node-fetch. [Accessed 22 March 2023].

[13] ts-node, “ts-node - npm,” [Online]. Available: https://www.npmjs.com/package/ts-node. [Accessed 22 March 2023].

[14] crypto-js, “crypto-js - npm,” [Online]. Available: https://www.npmjs.com/package/crypto-js. [Accessed 22 March 2023].

[15] jsonwebtoken, “jsonwebtoken - npm,” [Online]. Available: https://www.npmjs.com/package/jsonwebtoken. [Accessed 22 March 2023].

[16] method-override, “method-override - npm,” [Online]. Available: https://www.npmjs.com/package/method-override. [Accessed 22 March 2023].

[17] mysql2, “mysql2 - npm,” [Online]. Available: https://www.npmjs.com/package/mysql2. [Accessed 22 March 2023].

[18] validator, “validator - npm,” [Online]. Available: https://www.npmjs.com/package/validator. [Accessed 22 March 2023].

[19] dotenv, “dotenv - npm,” [Online]. Available: https://www.npmjs.com/package/dotenv. [Accessed 20 December 2022].

[20] “npm @Types organization,” [Online]. Available: https://www.npmjs.com/~types. [Accessed 23 March 2023].

[21] “nodemon - npm,” [Online]. Available: https://www.npmjs.com/package/nodemon. [Accessed 23 March 2023].

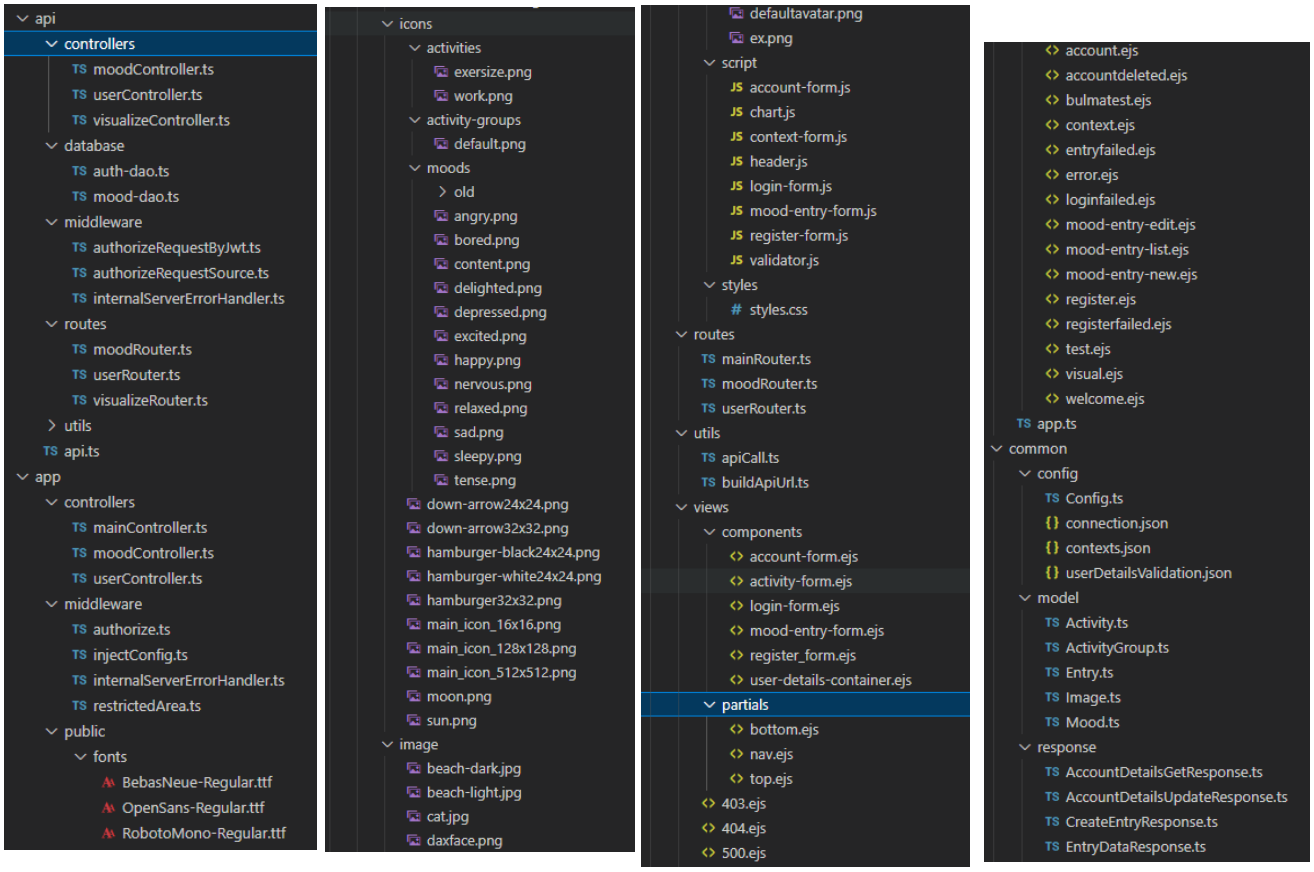
[22] ExpressJS, “ExpressJS Locals,” ExpressJS, [Online]. Available: https://expressjs.com/en/4x/api.html#res.locals. [Accessed 19 March 2023].

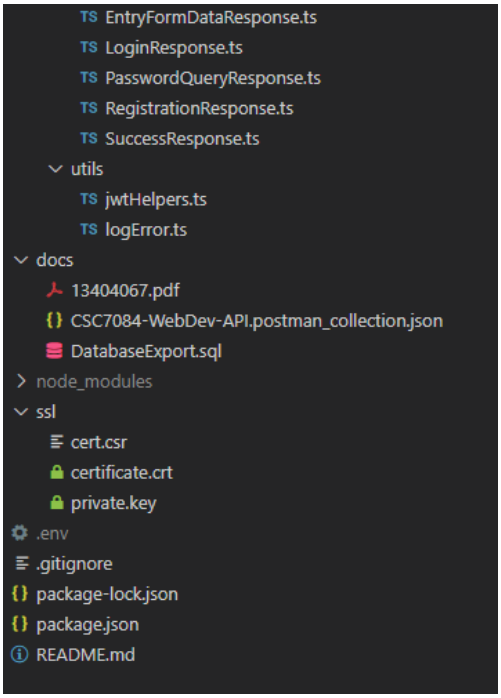
[23] P. Leach, M. Mealling and R. Salz, “RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace,” Internet Engineering Task Force, 17 January 2017. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4122. [Accessed 19 March 2023].

[24] Mozilla Developer Network, “HTTP request methods - HTTP | MDN,” Mozilla, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods. [Accessed 1 February 2023].

[25] Chart.js, “Chart.js | Open source HTML5 Charts for your website,” [Online]. Available: https://www.chartjs.org/. [Accessed 22 March 2023].

Appendix 2 – File Structure as visible in VSCode





The file list begins top left, and ends bottom left, to be read from left to right.