[Moods and Triggers 2](#_Toc129289427)

[Authentication & Security 3](#_Toc129289428)

[Login 3](#_Toc129289429)

[Authenticated Users & Restricted Areas 4](#_Toc129289430)

[API Authentication 5](#_Toc129289431)

# Moods and Triggers

Diagram

Description automatically generatedI 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 the its extra complexity.

Another strong contender was the Emotional State Model - which attempts to enumerate the discrete emotional states such as Happy, Angry etc. This would fit well within the constraints of this project, but it seemed overly simplistic, and quite a lot older than the other two models I considered.

Russell’s Circumplex Model of Affect was particularly attractive as it allowed 2 axes (valence and arousal) and 4 categories with which to classify each “mood” - see Figure 1. This would be invaluable during visualization of mood logs. I distilled this model into 12 separate moods, each have an arousal and valence property.

For mood triggers I used activities the user engaged in – those that prompted the mood – and a free-text *notes* field would be appropriate. 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*. This will be demonstrated in the video report.

~~When a user engages in mood logging, they~~ **~~(1)~~** ~~Choose a mood valence (positive/ negative) and are shown all the moods within that valence~~ **~~(2)~~** ~~choose a specific mood then optionally~~ **~~(3)~~** ~~enter some text notes about the entry and~~ **~~(4)~~** ~~Select an activity – or multiple activities. Only the selected mood is mandatory, all other information is optional. Each mood entry is also assigned a date & time stamp when it is created in the database. The timestamp and mood itself are not editable after the fact, only the notes and activities can be changed.~~

**STATUS AND VISUALISATION CRAIC HERE**

# Requirements

* **Validation**
  + **Client**
    - **How**
    - **Why**
    - **How it looks**
  + **Server**
    - **How**
    - **Why**
    - **How it looks**
* **Config**

**User Registration:** When navigating through the website whilst not logged in, the user may click on a hamburger button [3] in the top right corner. This will display a modal that contains the login form, and a link to the registration page. The user clicks on the registration link and a registration form is displayed.

This form’s inputs are validated both client-side and server-side. When the user begins to enter data in any of the 4 inputs, keyup events are fired and caught by an attached event handler function named handleValidateInput - defined in ./app/public/script/validator.js – This method takes the input element and validation message div CSS selectors and a validation call-back function as parameters, and so is reusable for validation of all input fields. Additionally, the password and password-confirm inputs are also validated with a second attached event handler function named handleValidatePasswordInputsTogether (also from validator.js), which also validates the password and confirm password input fields contain the same value.

The server-side injects validation configuration data into the client-side by passing config data to the rendering engine via ExpressJS locals [4], which then stores required config data in client-side JavaScript variables in a script tag in the ./app/views/partials/top.ejs file’s EJS/ HTML <header> tags. The validation call-back functions are also embedded into the client-side in this <header> tag (from ./app/public/script/validator.js), for use throughout client-side scripts. This top.ejs file is one of the layout template partials and is loaded for every page, which makes the specified config data and validator functions available throughout the client-side.

Config data is defined by a configuration class named Config – defined in ./common/config/Config.ts - that reads data from JSON configuration files. This Config class is imported in various files throughout the app/ api (and partially passed to client-side as discussed above). 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.

The validation message div’s for each input are populated from this configuration data (description discussed above) when the page is loaded, and a custom hidden CSS class is used to hide it. The keyup event handler method handleValidateInput calls the validator call-back function on the input’s value property. The call-back function uses the regex specified in the injected configuration data to validate the field’s value. If deemed valid, it adds a valid CSS / removes the invalid CSS class, which styles it appropriately and applies the hidden class to that validation error message div; and the opposite if deemed invalid.

If a user bypasses the client-side validation (e.g. executing document.querySelector(‘#registration-form’).submit() in the console), then the server also validates all fields before the API is called. The validation regex is obtained by the server-side code from the same Config class discussed above.

If any form input data is deemed invalid, a 400 status is returned and registerfailed.ejs template is rendered, which has the login form component embedded, with an additional div displaying the error messages returned from the server. If all form input data is deemed valid, then a request is sent to the API’s api/user/register POST endpoint. This endpoint is handled by the ./api/route/userRouter.ts ExpressJS Router which calls the .api/controllers/userController.ts’s register() method. I did server-side validation again on the API side, which is perhaps redundant, but I found it difficult to decide between server-side validation on the web app or the API side, so I erred on the side of caution and put input validation in both.

This register() method then calls the register instance method on the mood Data Access Object (DAO – these are defined in .api/database/\*-dao.ts). This DAO handles all database operations for mood related activities. For registration the following operations are undertaken: (1) insert new user into tbl\_user and retrieves tbl\_user .user\_id attribute from new record, (2) inserts *Default* activity group into tbl\_activity\_group – and related to the user’s user\_id, and (3) inserts default activities in this group (*Work* and *Exersize*) – also related to the user’s user\_id. Each user has this data seeded so that they may latter add, remove and edit these contextual activities.

After database operations are complete a JSON response is returned to the web application (modelled by the ./common/response/RegistrationResponse.ts class). This JSON response indicates if the action was successful, any errors, if there are any as a string[]. If the API’s registration response indicates it is successful, the next route specific middleware function is called (controller.attemptLogin - discussed in next section), which logs the new user in.

If the registration was unsuccessful then the registerfailed.ejs template is rendered with the error strings returned from the API displayed above the form.

User input is checked at client, web app and API levels.:

* Client side the regular expression checks are used to enforce safe characters only. these regular expression checks enforce only safe characters are present.
* If these are bypassed input also sanitized server side via the same regex. As a precaution to the regex being changed into the future and introducing a vulnerability input is passed to the JavaScript native method encodeURIComponent(string) to escape all HTML characters. Its return is used to create the HTTP request’s x-www-form-encoded parameters destined for the API.
* The regex check is a

**DATABASE ENCRYPTION CRAIC HERE?**

**User Login:** The user logs in via clicking on the hamburger button [3] which displays the login form. This form also has client-side and server-side validation. This form employs the exact same method for client-side validation as the registration form (inject validation data from Config class on the server-side, store relevant config on client-side, validates using handleValidateInput event handler, which utilizes this validation config data, display error in an absolutely positioned div attached to input).

If the client-side validation deems the request valid, or client-side validation is bypassed by the user, then a call is made to the /api/user/login POST endpoint. Much like user registration, the web application’s userRouter receives the request and this time calls the controller.attemptLogin method. This method first validates the login form’s username and password inputs exist. If they do not a 400 with appropriate error messages, which are then passed to the loginfailed.ejs template where they are displayed to the user.

Then the apiCall() method from ./app/utils/apiCall.ts a method my web applications uses that abtracts the actual API call itself into

This protects against bypassing client-side protection of form validation.

If

# Authentication & Security

## 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 2 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 POST request to the API is authorized via an API key. 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 method of my AuthApiDataAccessObject class – 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 3 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, and the response contains a token property, the token is retrieved and stored in a cookie on the user’s machine named token.

If the success property is false, the loginfailed.ejs template is rendered with a 401 status code. If the success property is true and the token property is undefined, or otherwise falsy, the user is redirected to the /500 route in the fallbackRouter (with a status also set to 500). This indicates to the user that an error has occurred on the server.

This describes the basic method by which a user is authenticated.

## Authenticated Users & Restricted Areas

When a user attempts to access any part of the web site, a middleware function named authenticate then attempts to the read the JWT from the token cookie and verify it. It does so by attempting to decode the token into a plain JSON object using a secret key. As cryptography is an area of utmost importance, and the technical level required to implement it safely is relatively high, I decided to install and import the jsonwebtoken node module. This gives us access to the verify() and sign() methods, which can be used to verify/decode and create JWTs respectively. I’ve wrapped the use of these two functions in my own custom jwtHelpers.ts module, so that changing the library used to sign and verify tokens can be done relatively easily – by editing this single module.

|  |
| --- |
| function verifyToken(token: string): JwtPayload {    return jwt.verify(token, process.env.MOODR\_TOKEN\_SECRET!)  as JwtPayload;  } |
| Figure 4 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 this jwtHelpers.ts module via the verifyToken() (figure x) and createToken() methods. This module is imported by both the web app (in the middleware authenticate) and the API (in the middleware authenticateRequestByJwt and in the class AuthApiDataAccessObject), allowing both to process these JWTs in the same manner. In reality both the API and web app would be deployed with the secret in a configuration file, and not be so tightly coupled to a utility method directly like this.

In the authenticate middleware An ExpressJS local variable inside the ExpressJS Response object is declared, if the token was successfully verified, it is initialised to true, and otherwise to false. This expressJS local boolean serves as the flag by which the site may decide to permit access to a given route (note all variables assigned as properties of the res.locals object are then accessible in the template files. res.locals.example set in a middleware function is then accessible as example in a template rendered by that route). All authentication-restricted routes call another middleware function named restrictedArea, that checks this flag, and if it is falsy (e.g., null, undefined or false - among others), then the user is redirected to a /forbidden route, that simply informs the user this route is forbidden without a successful login.

Using this methodology, my web application’s login-restricted routes can allow access to authenticated users only.

## API Authentication

The API has a primary method of authorization for requests originating from the web application’s server/ back-end and another secondary method for HTTP requests that are originate from client-side script charts.js.

The primary method of authorization is via that of an API key set in the Authorization header.

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 5 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.

# REST API Implementation

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.

Graphical user interface, text, application, website

Description automatically generatedThis 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.

Figure 6 API keys stored in tbl\_key

~~The keys are passed via the Authorization header of requests they receive. They are randomly created unique (enforced via uniqueness constraint in the table) GUIDs [5] stored in the tbl\_key table of the database (Figure 6). They can be easily disabled by changing the active bit(1) attribute to 0. The API’s ExpressJS middleware function that authorizes the API key checks for this flag when validating the key. The API key in use by the web application is stored in an environment variable (more specifically it is injected into the environment variables from a .env file using the dotenv node module).~~

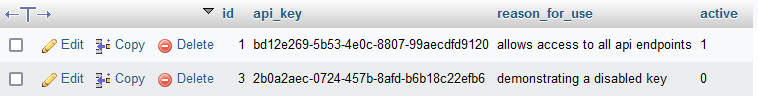
## Endpoints

A list of all API endpoints follows, with any parameters replaced with their type as so <<number>>. All endpoints accept application/x-www-form-encoded content type in request bodies. All endpoints use the building express Response method .json() [6] to convert from TypeScript response objects to JSON via JSON.stringify(). These response objects, discussed below, are stored in the ./common/response/ folder.

|  |  |  |
| --- | --- | --- |
| **Route** | **Method** | **Description** |
| **Router: ./api/routes/userRouter.ts** | | |
| /api/user/login | POST | Expects an x-www-url-encoded POST body containing username and password URL encoded key/ value pairs, and returns a response modelled by LoginResponse.ts. If post body is missing data, or if the passed username or password fail their regex tests from Config as discussed above, returns a 400 with an unsuccessful message. If username and password are correct a login JWT is created and returned, where it is set in the “token” cookie. If the username and password combination is incorrect a 401 is returned. |
| /api/user/register | POST |  |
| /api/user/<<user ID>> | GET |  |
| PATCH |  |
| DELETE |  |
| /api/user/<<user ID>>/password | PATCH |  |
| Router: ./api/routes/moodRouter.ts | | |
| /api/mood/<<user ID>>/new | GET | Get the form data required to display the mood logging form/ template. |
| POST | Create a new mood log/ entry. |
| /api/mood/<<user ID>>/list | GET | Get all the mood logs for a particular user, processed and sorted into days, chronologically in descending order. |
| /api/mood/<<user ID>>/<<entry ID>> | GET | Gets the a specific mood log’s data, required to display the edit log page. |
| PUT | Update a mood log entries contextual information (notes and activities). I went with PUT instead of PATCH as I am updating all allowable attributes of the resource at once. |
| DELETE | Delete a mood log entry |

# REST API

All database operations performed by the web application are done so through a RESTful API which is also an ExpressJS application. It is divided into 3 ExpressJS Routers - *authRouter, moodRouter* and *visualizeRouter*, to maintain separation of concerns. As discussed in the [Security](#_Authentication_&_Security) section, requests to the *authRouter* and *moodRouter* are authorized via API keys and the *visualizeRouter* is authorized via JWT– both sent as Authorization headers.

 Currently my database has a table named *tbl\_key* (Figure 6), in which are stored API keys that the API will accept. New API keys can be added simply by inserting into this table. Old/ current keys can be deactivated by simply updating the active bit(1) attribute to be 0. The code in the API that checks the Authorization header for this API key can be seen in Figure 6. The API retrieves the Authorization header prior to this line, and forms a prepared SQL statement which checks both the key exists, and is currently active. The *format()* function is provided by the mysql2 node module/ library, and allows us to prepare SQL statement with escaped strings, to prevent SQL injection attacks. In this manner a key may be deactivated once compromised, or of a certain age.

|  |
| --- |
| const sql = format("SELECT COUNT(\*) AS `count` FROM tbl\_key k WHERE k.api\_key=? AND k.active=1", [authHeader]);  // evaluates to  // SELECT COUNT(\*) AS `count` FROM tbl\_key k WHERE k.api\_key='bd12e269-5b53-4e0c-8807-99aecdfd9120' AND k.active=1 |

Table

Description automatically generated

Figure 7 Current API Keys

Figure 8 API authentication code

Figure 9 API Check code

The database is then queried and if the key does not exist, or is not active (*count* returned is 0) then the API key is rejected, and a 401 Not Authorized status is returned with a request body modelled by my *SuccessResponse* class.

Figure 10 API Keys storage in DB

|  |
| --- |
| {      "success": false,      "errors": [          "You are not authorized."      ]  }  Figure 11 Not Authorized SuccessResponse |

If the key does exist, and is active (*count* returned is not 0), then the *next()* Express routing callback function is invoked, which moves the now authenticated request onto the next relevant route.

# 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] | “Hamburger button,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Hamburger\_button. [Accessed 03 March 2023]. |