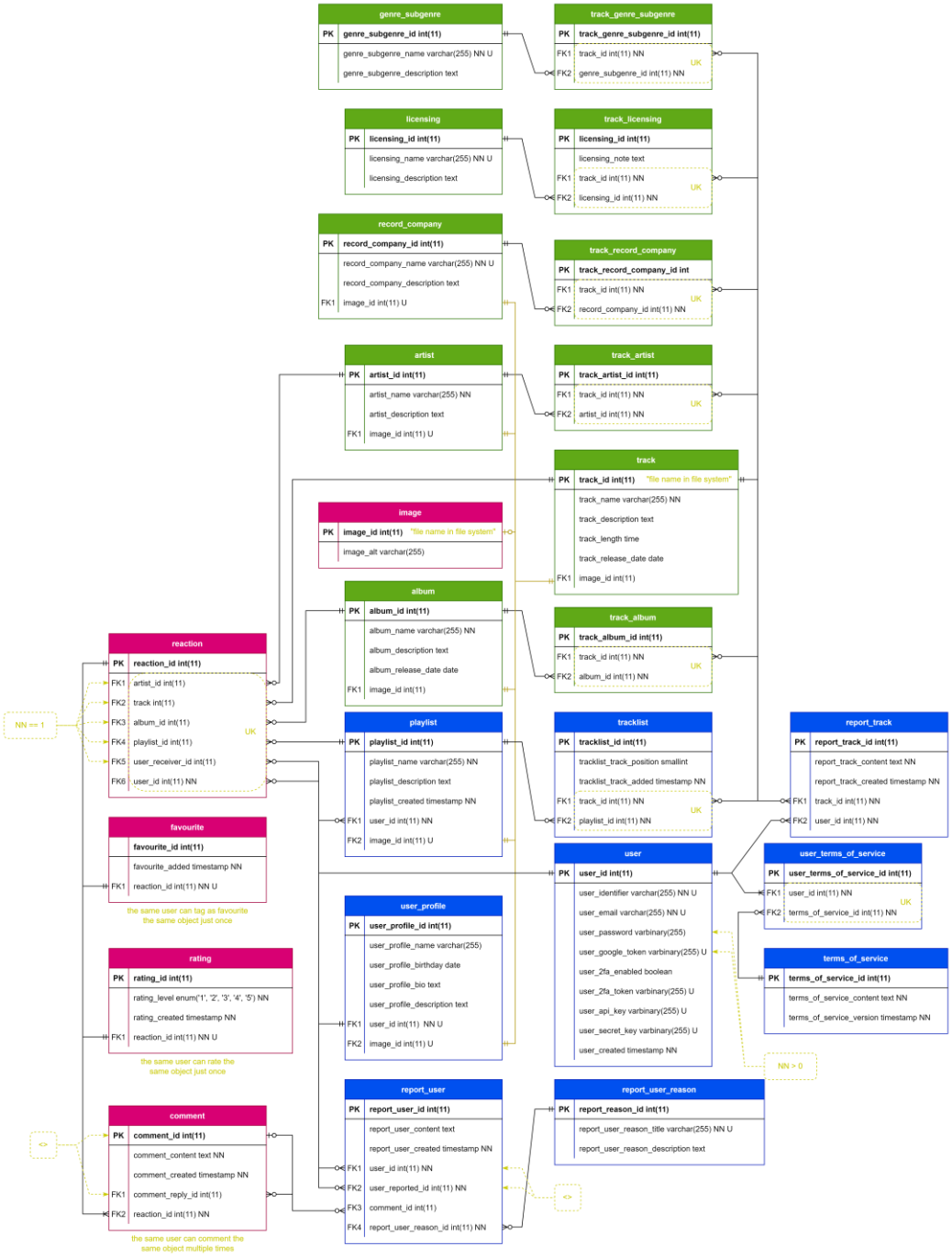
Stacksofwax

Roberto Lo Duca 40386172

Please consider that English is not my native language.

Entity Relationship (ER) diagram



High-level description of system developed

The system allows registered, non-registered users to filter tracks, albums, artists, users and playlists created by users. It's also possible for those elements to be commented on, rated and added to favorites by registered users. Registered users should be able also to comment, reply and report other users' comments if necessary. Each comment is associated with the element in the database that the user wishes to comment on. For example, a registered user can leave a comment on the artist page and leave a review. He can also add the artist as favorite. The website should allow users to see all the comments for each element and an average rating will be calculated. The website is meant to create a social environment for music lovers and be a social network for music in general. It also allows you to listen to new tracks and add your favorite tracks to a playlist. Every user on the website can filter, and search playlists created by other users and comment/review them. A playlist has a description, creation date, position for ordering each element, an image that the user can set etc.

Each track has information about the album if there's one, artists, creation date, description, record company, licensing, genre, subgenre etc. All this information can be filtered in the dataset to retrieve a particular track in the database. Users can filter tracks based on all these elements, but also other elements can be filtered as well in the same way.

Just like Spotify, our website doesn't allow users to add new tracks, but instead they are added by the administrators of the system by using other services. Artists or record labels need to obtain the necessary rights to be uploaded to our website. They can use a music distribution platform, such as DistroKid, TuneCore, CD Baby, or similar services, to upload their songs to Spotify. These platforms offer services for distributing music to streaming platforms. In the same way our system allows new tracks to be added, to avoid misleading information and wrong descriptions. Anyway, to avoid any errors we have a system to store the reports of wrong information by the community to allow admins to fix the wrong data. Each track can have more than one genre, licensing type, record company and artist, and it could be present in more than one album. For example, a song may fall into multiple genres and have multiple license types such as "Creative Commons" and "All Rights Reserved". In our system, a song could be classified as both Alternative Rock, Pop or Electronic and Ambient. Users can find their favorite songs by using this data, allowing for hierarchical relationships between genres and sub-genres. A genre can also be a sub-genre.

The website is designed to allow scalability, performance, and security. It is meant to be a web app with a fully structured API system, to allow users to use the data for example in projects like a discord bot for music, a telegram bot etc. There are limits on the use of API in our service that can be unlocked under certain conditions. Like a paid subscription, especially if the APIs are used for commercial purposes. Our system employs advanced cryptographic systems, such as AES encryption with IV (Initialization Vector) for securely transmitting sensitive data like passwords through POST requests, adding an additional layer of security to protect user information from potential threats.

The level of security in our system avoids malicious attacks like dos attacks, by limiting each request with a weight for each endpoint, and eventually ban the IP, replay attack, also known as a playback attack, protecting user information with hashing and AES encryption, signed communication with HMAC, signature validation etc. Developers can generate API keys and secret keys through the platform's secure interface, which grants them access to specific functionalities while maintaining strict security protocols. Every user should accept the terms of service before using our platform.

Brief analysis of development approach

I developed the system by starting from a well-structured database. I started from the track table that I connected with many to many relationships to all the other elements. Each track can have many artists, genre, subgenre, licensing, record companies and can be present in more albums and playlists. I forced uniqueness in some elements to ensure to avoid repetition of data. An album can be empty to allow administrators to fill it with data in a second moment after the creation, but for other elements in the database I forced with some triggers the elimination of elements to reduce the SQL query in the code itself. By eliminating or minimizing data redundancy and adhering to appropriate database normalization rules, I improved overall data integrity and performance. Each track, like many other elements in the database, relates to the image table. Both image and tracks are heavy data elements, so I decided to store them in the file system. Storing binary files in a file system offers better performance compared to storing them directly in a database. File systems are optimized for handling large files and can provide faster read/write operations, especially for large binary files such as songs and photos. To retrieve the image, I use the image id associated with the element. Every time an element adds an image in the database the id count increase and a new file is added in the file system.

The table for users contains data that can be added by the user. For these I included the possibility for the user to create playlists, report tracks, report other users and associate the report to a comment if the report is about a comment and not the user profile itself and I added a list of reasons that the user can choose. Furthermore, the user has total control on the social aspect of the website, by commenting, adding favorites and reviewing any of the elements of the dataset. For example, he can leave a comment to an artist, a track, an album, playlist and other users. To do so I had to use a table called reaction that allows me to connect each one of these “reactions” to his element. To keep the dataset consistent, I forced some rules in the code itself. For the reaction table I used a CHECK that counts between the elements that are not null, if more than one element is not null an error will prevent the creation of the table. If the reaction is removed the data will be also removed with an ON DELETE CASCADE. For the playlist, the user needs to create empty ones in order to fill them in a second moment. He will be able on the website after the filtering of the track to add the track in one of the playlists he created if he’s logged in. If a playlist is removed a trigger will also remove the image connected with it, same for the user, after deleting the user all the images connected with all the playlists of the user will be deleted with the playlists. These elements are personal to the user, and they do not need to stay in the system if the user leaves. Another example is with the reaction table. If either the playlist or the user that commented on that playlist is deleted, all the reactions of that user or playlist, connected with all the different reactions (favorite rating, comments, replies to that comment) will be automatically destroyed. The whole system is designed to be fully consistent and doesn’t allow unused data to exist. Not all the data connected with the user will be deleted anyway, for example reports. There’s coherency in the data, for example a comment cannot reply to itself.

Each user has a user_profile. I divided these tables to separate private and mandatory data. Public data can be seen and reacted to by other users. The dataset allows users to log in to the system by using different systems. He could generate a pair of API keys to use the API, (the API server fully supports already this authentication), google tokens to authenticate with google, enable 2fa to improve security, and have his password stored with a hashed salted system and unique identifiers to allow other users to find them in the system without using the id or email (privacy reasons). The user identifier is the only public element from the user table and can be seen by other users only if they find the user itself by his

user_profile_name. A term of service table is added to collect versions of them and to keep track of which term of service version the user agreed with.

No decisions were taken without a valid reason: For example, the choice of not adding a total from the review (ENUM between 1 and 5) of all users in all the elements is to avoid repetition of data and modification each time a new user adds a review, otherwise the system could be more complex to maintain. Instead, the system will never modify the review in the element but will just recalculate it when required. That's why in the UI the reviews change dynamically every time a new one is added. The calculation is made at the database level with a query, before the tracks are extracted and added in a json format. This choice was taken to avoid collecting in the API server all the reviews, because they can be millions and could not fit in the system memory. Even if they fit in the memory or if they are collected and calculated in chunks, there's a big transfer of data from the database to the API that will slow down the system, same concept for filtering[i]. To show a list of items bigger than 500 for example, I need the user to select the page number and one of the allowed sizes. If the website allows the user to select a bigger chunk of data the server will slow down, eventually run "out of memory". I was inspired by the coinmarketcap[ii] data.

I started to build the framework with express[iii] and a solid structure of my API and web app server simultaneously. I used as an example the Binance API[iv]. Binance as a cryptocurrency exchange uses really advanced security protocols and I was inspired by it. I just checked the documentation and decided to build my API for developers in a similar way. My server API has weighted limits that can be set for every endpoint differently, as Binance does, at the IP level. If the request comes from the web app server, the limit will be skipped. There is a library to set calls limits for every IP, but it's not weighted, and it does not allow me to set specific rules, so I decided to build it by myself and add it as a middleware called limit_request. Even if the user is not allowed on the website he can spam calls to the server, so after the weight is reached the weight count does not stop. The server returns to the user a json with 429, with the message "too many requests" and if reach a certain level of spam It returns a 403 "forbidden status" with a message that tells when the ban will end. Before each ban, the developer can check the Retry-After header that tells how many ms the user needs to wait to use my service again and avoid the ban (I believe that it is very handy to have). Each interval of time the weights of all the IP will be settled to 0. (No code was copied from binance but only the concept).To generate the API key and secret key I used the library crypto[v]. Crypto generates a series of random bites and I can choose the length. Then I converted them into hexadecimal and stored them in the environment variables. This allows me to access them from both the servers. I used the fs[vi] library to edit the .env file where I store the pair. I used dotenv[vii] to store the environment variables every time the server starts, I did the same for the mysql login variables. I modified the nodemon[viii] configuration in package.json to restart the server every time the .env is modified. In this way every time the API server starts, modifies the .env of the web app server and restarts it with a new key pair. The database communication is done with mysql[ix] library with a limit of 100 connections and 1000 queues, to handle the hardware limitations of the server. There's another middleware for limiting the response called limit_response. It accepts and modifies the query called quantity before reaching the endpoint. 400 will be returned with the specific error message and the maximum limit will be applied if undefined.

Some endpoints can be accessed only by the web app server, like the login and register one. If the endpoint should be limited, a middleware is added after the url, with a boolean passed. This middleware I built called web_app_request checks if the request comes from the web app or any other authorized

app by checking if the API key is valid and if the message was encrypted with the same secret key. Both servers store the same pair of keys, and they are generated after each reboot, so there's no need to store them in a database like for the users. If the boolean is set to false, the message does not need to be decrypted. In that situation it will just check if the API key is allowed and the hmac_autentication middleware will check if the origin of the message is safe. If true means that some secured data needs to be decrypted from the body of the POST request used in logins and database modifications. If the communication is https is not needed but it adds a layer of security to the sensible data of the user. The code for the strong encryption was taken from siwalikm[x] github and modified accordingly. I chose AES 256 with initialization vector (the random IV generates different encrypted result for every encryption), one of the strongest encryption systems, I chose 256 bits because the length of my API key pairs is 256 bit (32 byte). I used again crypto for encryption and decryption in both servers.

For developers and web app shared endpoints I use the same system used in binance, HMAC with sha256. I found the code for node in jasny[xi] github, with crypto library. I created the middleware where I check the API key first from the web app, then for all the users in the database. Then I check if the hashed message was generated with the same secret key. If the hash matches the user or web app or any other admin is allowed to use the API shared endpoints. registration must contain a timestamp and must be at the right timeframe to allow the request. In this way we block the attacker to intercept and retransmit a valid message to our server, also known as reply attack. When the user is validated a query with the user id is added, or in the case of the web app a boolean admin true. All the requests use this middleware, except for login and registration that have a different validation process. If we try to access any of the endpoints, and we do not include an API key in the header or the signature with the timestamp in the query a 400 bad request will be sent, and the relative message. Tested with Postman.

An example of how the website works: The user registers by compiling the form. If any of the fields are wrong a message will communicate the errors to the user. After sending email and password a middleware in the web app called check_credential, will check with validator[xii] if any of the fields are correctly formatted. Another library called zxcvbn[xiii] will check dynamically how strong the password is and show a bar. Values are passed to the web server in the body (used to store protected data) that will encrypt (for some endpoints) and uses axios[xiv] to communicate with the API. The API decrypts, checks the timestamp and stores the password if the email and identifier are unique, and hashes it with salt by using bcrypt[xv] to prevent legal consequences in case of a fallacy of the system. The API will send a 200 and redirect to login. After logging in with a similar system and using bcrypt again to confront the password, an express-session[xvi] will be opened in the web app, using the cookie-parser[xvii] middleware, and user info like the image will be saved on it. The session has a time-lapse of 1h. If the user does not have an image, a random one is generated by using gravatar API[xviii]. I use path[xix] library to add the public and database folder on my project to retrieve images and tracks. All registered and non, users, can then filter tracks from the database with a form. In this case the request is a get and all the data is passed as query to allow the user to send the filtering with a link to other users. Then the data is modified by the hmac_signature function. It adds to the query a timestamp and generates a hash with sha256 of the whole message, by using the web app secret key before calling the API. Secret keys should never be sent after they are shown to the user, and if they are lost, they need to be destroyed and regenerated randomly. For the user interface I used bootstrap[xx].

Instructions on how to run the system (same info in README.txt)

I used MySQL with a collation utf8mb4_unicode_ci that allows case insensitive and supports a wide range of language sets. Considering the specific filtering requirements, other collations may not be suitable.

To run the system locally I used phpMyAdmin on a windows machine. I used XAMP (xampp-windows-x64-8.2.0-0-VS16-installer.exe). I created a database called 40386172 (utf8mb4_unicode_ci) and imported the file from:

.\stacksofwax\database\40386172.sql

To use different names, port or credentials to start the local database server, the configuration file is located in:

.\stacksofwax\apiserver\.env

The web app server code is located in

.\stacksofwax\apiserver

The API server code is located in

.\stacksofwax\webserver

To run the system, open both folders with the servers in vscode or in my case in vscodium (open-source version of vscode). Run both, the order doesn't matter because if the api server runs before the web server, it generates the environment API keys that modify the webserver .env file. The webserver should automatically restart when the pair is generated, if doesn't restart it manually or start the apiserver first.To start the servers, use the command "npx nodemon".

After starting both servers, go to http://localhost:3000/tracks and navigate from there to the dashboard and other pages. There are 2 users registered, they both reviewed 1 track called "Cracks". If you want to see the average result of all the reviews, you can find the track Cracks in track name and see the review.

The email and password to login with the 2 registered accounts are:

Email: robertoloduca@hotmail.it

Password: aaaaaaaaaA.1

Email: aliceloduca@hotmail.it

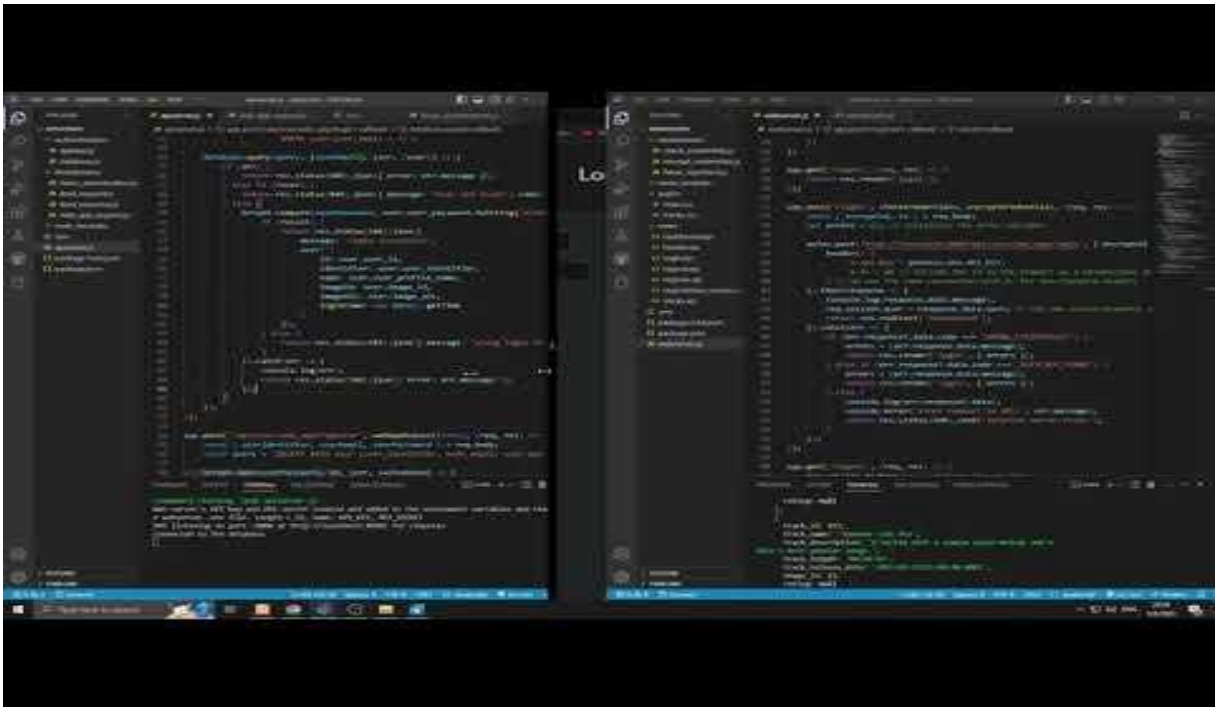
Password: aaaaaaaaaA.1

in the folder .\stacksofwax\documentation there is an example of tracks stored in my database and the file in pdf and draw.io version of the database structure. please refer to them to see in a better resolution the ER diagram:

.\stacksofwax\documentation\ER stacksofwax.drawio.pdf

Video demonstration link

[Stacksofwax]



REFERENCES TO EXTERNAL CODE SNIPPETS, LIBRARIES, DEPENDENCIES

[i] https://stackoverflow.com/questions/52346685/filters-logic-should-be-on-frontend-or-backend
[ii] https://coinmarketcap.com/

[iii] exprress library
[iv] https://www.binance.com/en/binance-api

[v] crypto library
[vi] fs library
[vii] dotenv library https://www.npmjs.com/package/dotenv
[viii] nodemon library
[ix] mysql library
[x] https://gist.github.com/siwalikm/8311cf0a287b98ef67c73c1b03b47154
[xi] https://gist.github.com/jasny/2200f68f8109b22e61863466374a5c1d

[xii] validator library https://www.npmjs.com/package/validator
[xiii] zxcvbn library https://cdnjs.com/libraries/zxcvbn
[xiv] axios library
[xv] bcrypt library https://www.npmjs.com/package/bcrypt
[xvi] express-session library
[xvii] cookie-parser library
[xviii] gravatar API https://www.gravatar.com/avatar/3678832349057?d=identicon
[xix] path library
[xx] bootstrap framework https://getbootstrap.com/