Vinyl Data API Web Server Documentation

GitHub Repo

Trello Board

ERD

Table of Contents

Installation and Setup

R1 / R2: Problem Identification and Justification

The problem that this app is trying to solve lies with people who have large collections of vinyl records. Whether they are vinyl collectors who simply love to listen to records at home, or vinyl DJs who cart around many crates full of records to use in their live performances. As you can imagine, digital music is already catalogued by applications such as iTunes, Spotify or Rekordbox and users of the apps can easily create playlists and group their albums/tracks in many different ways for listening and performing. Obviously, vinyl listeners don't have this luxury and are forced to dig through their hefty collections to find what they're looking for and manually manage/sort their records on the shelf and in their crates to create some sort of system of grouping and searching for records with similar attributes.

For Vinyl DJs in particular; a lot of the time, the vinyl (particularly the older vinyl) will not provide all of the information on the record sleeve that the DJ needs in order to mix the record into the set effectively. This information includes:

1. Tempo or BPM of each track - eg: 132BPM
2. 33, 45 or 78 RPM (rotations per minute)
3. Key (eg: A Minor)

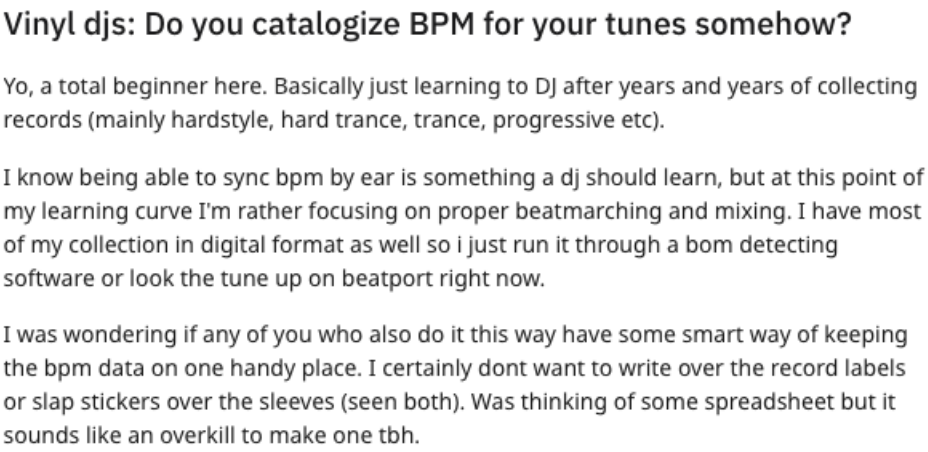
As a listener of vinyl records, you probably aren't as fussed about the BPM and key of the tracks as the vinyl DJs are (although some might be), but having a comprehensive and easily accessible database full of the names/ artists/ tracks and especially RPM (rotations per minute) would be a very helpful tool to have to organize the data of your record collection. I would imagine that people with very large collections would not only find it difficult to locate an album in their collection but to even remember which records they have collected, to begin with.

The following images are screenshots taken from Reddit, showing the need for this type of vinyl data organization:

Reddit screenshot 1:

(shoyei, 2019)

Reddit screenshot 2:



Vinyl djs: Do you catalogize BPM for your tunes somehow?

Yo, a total beginner here. Basically just learning to DJ after years and years of collecting records (mainly hardstyle, hard trance, trance, progressive etc).

I know being able to sync bpm by ear is something a dj should learn, but at this point of my learning curve I'm rather focusing on proper beatmarching and mixing. I have most of my collection in digital format as well so i just run it through a bom detecting software or look the tune up on beatport right now.

I was wondering if any of you who also do it this way have some smart way of keeping the bpm data on one handy place. I certainly dont want to write over the record labels or slap stickers over the sleeves (seen both). Was thinking of some spreadsheet but it sounds like an overkill to make one tbh.

(ceeroSVK, 2022)

This API is designed to assist **vinyl DJs** when performing live in the following ways:

- The DJ will not have to pull out vinyl from the crates and try and read the sleeve for information about the records or specific tracks on the records before using them to mix.
- Each vinyl will have an ID that the user can search via the database in order to retrieve the data on the records/ tracks before digging through the crates (which can often be in a dark space).
- The user will be able to update their record collection with full CRUD capabilities: create, read, update and delete records and tracks.
- The database will be filled with the user's specific vinyl collection without the need to sift through any records/tracks that aren’t in their own collection. (Eg: if you were to use the internet or Rekordbox to retrieve this information).

This API is designed to assist **vinyl collectors/listeners** in the following ways:

- The user will not have to flick through their extensive collection, whether it’s a shelf in their living room full of vinyl or a box sitting in the garage.
- The user will be able to locate any records with a specific attribute, for example: any records with an RPM of 45 and so on..
- The user can easily see a list of their entire collection if they simply wanted to browse for anything non-specific.
- The user might be thinking of the name of a track they want to listen to but can’t remember which album it’s on or even if they own the record.

In summary, the objective of this project is to help users of any kind store and manage their vinyl record data and to create a way for anyone with large vinyl collections to search for specific attributes on their records. I imagine people running record stores would use some sort of software similar to this to track the records kept in their stores, but for the sake of the project, it's aimed at private users with large collections.

R3: Justification of the Database System

There are a wide variety of database management systems that exist, each possessing unique pros and cons that cater to a diverse range of scenarios. In order to guarantee the selection of the most suitable database management system for this project, an evaluation was performed on alternative options.

PostgreSQL is a popular relational database management system that stores data in tables with predefined relationships. In contrast, a popular non-relational database management system like MongoDB stores data in collections with flexible schemas.

Benefits of PostgreSQL as a relational database management system include:

- ACID compliance, ensuring consistency and reliability of data.
- Powerful query capabilities with support for advanced data types like arrays, JSON, and XML.
- Ability to handle large volumes of data with efficient indexing and partitioning.
- Postgres allows you to create custom functions and operators, which makes it far easier to add new features and functionality to the database.

Some potential drawbacks of PostgreSQL include:

- Limited scalability due to its reliance on predefined relationships between tables.
- Higher maintenance requirements for ensuring optimal performance.

Benefits of MongoDB as a non-relational database management system include:

- Flexibility in schema design, allowing for easier adaptation to changing data requirements.
- Scalability for handling large amounts of unstructured or semi-structured data.
- Support for distributed databases for increased fault tolerance and performance.

Some potential drawbacks of MongoDB include:

- No ACID compliance, meaning that data consistency and reliability may not be guaranteed in all scenarios.
- Less powerful query capabilities compared to PostgreSQL for complex queries and analysis.

An RDMS simplifies the process of categorizing data into distinct entities and establishing relationships between them in an efficient manner. This allows for the structured organization of data and enables users to perform complex queries on the database with ease and speed.

Because of the size and nature of this application, PostgreSQL (a Relational Database Management System or RDMS) was chosen for a number of reasons which include:

ACID compliance. The term "ACID" represents four principles; Atomicity, Consistency, Isolation, and Durability. The principles are there to ensure the reliability of transactions within a database.

"The presence of four properties — atomicity, consistency, isolation and durability — can ensure that a database transaction is completed in a timely manner. When databases possess these properties, they are said to be ACID-compliant." (MariaDB, 2018)

For example, if a user adds a new vinyl record to their collection, an ACID-compliant database will ensure that the data is stored completely and accurately, without any errors or inconsistencies. Similarly, if a user updates or deletes a record, an ACID-compliant database will ensure that the changes are processed correctly and that no data is lost or corrupted. This is crucial for the longevity of this application.

As mentioned, other database systems such as MongoDB allow for extensive scalability whereas postgres has limited scalability options due to its reliance on predefined relationships between tables. This is ok here, because the relationships have been carefully considered before the production of the application and are planned to stay the way they are.

In this particular application, the use of an RDMS provides several advantages over non-relational databases. Since the data being stored has consistent attributes, the more rigid schema of an RDMS helps ensure domain integrity. Although non-relational databases offer greater flexibility, this advantage is not critical in this case, as the overall structure of the data across all tables in the database is not likely to change significantly over time.

R4: Functionalities and Benefits of an ORM

ORM, or Object-Relational Mapping, is a programming technique that establishes a connection between object-oriented programs and relational databases, typically through a bridge mechanism. In other words, an ORM can be thought of as that layer that links OOP (object-oriented programming) to the relational database.

In OOP languages, when working with databases, there are four primary operations that are performed to manipulate data. They are: create, read, update, and delete (CRUD). These operations are typically carried out in relational databases using SQL, as per its design.

Typically, queries using SQL are made to perform these actions on the data in a database. While this is perfectly acceptable and even required, ORM and ORM tools are there to facilitate an alternate method of interacting between the database and various OOP languages such as Python for a number of reasons.

"With ORM tools in place to manage the data interface, developers don't need to worry about building the perfect database schema beforehand." (Contributor, 2022)

To give an idea of how an ORM such as SQLAlchemy can be used to streamline queries and make them easier for a developer to implement, we'll take a look at an example from the code of this project.

The following is an example of SQL code that "gets" data about a record (vinyl record) from the database:

```
select * from "records" where id = 1;
```

The code returns data bout the record (with id = 1) stored in the database. In this example, the data will include record_id, album_title, rpm and user_id. Whereas, a tool in ORM can perform the same query in a different format.

```
Record.query.filter_by(id=id).first()
```

This allows for a few things. You can define an object to this line of code and return the "jsonified" version of this object to the browser or a tool such as Postman or Insomnia. You can build a function around this object and create a route that includes a GET method and takes the record_id as a parameter so that when searching for the specific record in the browser (or Postman/ Insomnia) you can filter out the other records in the table in order to retrieve the one you are looking for.

This way of working applies to all of the CRUD functionalities and will be implemented in all routes that are created in the controllers. To break that down, The ORM provides a high-level query language that enables developers to retrieve data from the database using object-oriented concepts. This query language typically translates to SQL statements that are executed against the database.

ORM also maps database tables to classes and objects in the programming language used. This mapping is often defined using metadata, such as annotations or configuration files. Mapping enables developers to work with the database using familiar object-oriented programming concepts.

"ORMs create a model of the object-oriented program with a high-level of abstraction. In other words, it makes a level of logic without the underlying details of the code. Mapping describes the relationship between an object and the data without knowing how the data is structured." (Liang, 2021)

ORM also provides mechanisms for modelling relationships between objects, such as one-to-one, one-to-many, and many-to-many relationships. It provides mechanisms for validating objects before they are persisted to the database. This can involve checking that objects meet certain criteria, such as having valid values for required fields, and that they do not violate any constraints or business rules.

The ORM also provides tools for managing the database schema, such as creating tables, modifying columns, and generating schema migrations. This simplifies the process of making changes to the database schema and ensures that the application's data model remains in sync with the database.

Benefits of using on ORM:

An ORM abstracts the underlying database operations, allowing developers to work with objects and classes in the programming language of their choice rather than writing SQL queries directly. This makes it easier to write and maintain code, especially for developers who are not as familiar with SQL or the specific database being used.

"Object-relational mapping tools help developers automate object-to-table and table-to-object data conversion while connecting a database to an application with minimum SQL knowledge. O/R mapping allows developers to overcome the challenges of writing and interpreting SQL code and instead focus on generating business logic to ensure higher productivity with lower development and maintenance costs." (Contributor, 2022)

Since an ORM provides a layer of abstraction between the application and the database, it makes the application platform-independent. This means that the application can be ported to a different database system with minimal changes to the code.

ORM tools can significantly reduce the amount of boilerplate code that developers have to write when working with databases, allowing them to focus on writing business logic instead. This can lead to faster development times and increased productivity.

"Manually writing data-access code is massively tedious work taking up much of developers' valuable time without adding much value to an application's functionality. Leveraging an O/R mapping tool helps developers significantly reduce development time by automatically generating the code." (Contributor, 2022)

ORMs can help prevent SQL injection attacks, a common type of attack where malicious SQL statements are inserted into user input. By using parameterized queries, an ORM can prevent these attacks and help ensure the security of the application.

"An effective ORM tool also ensures security for applications by shielding them from SQL injection attacks. The O/R mapping framework helps to filter the data to ensure robust safety for the developed applications." (Contributor, 2022)

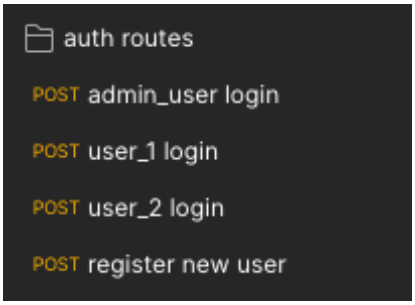
Since an ORM provides a high-level interface to the database, it can make code more maintainable by reducing the amount of low-level SQL code that developers have to manage. This can make it easier to refactor the code as needed and make changes to the database schema.

- Arguments:
- Description:
- Authentication:
- Headers-Authorizatrion: Bearer {Token} -
- Request Body:
- Request Response:

R5: API Endpoints

Auth Routes

Auth Routes in Postman



/auth/login

Method: POST

- Arguments: None
- Description: A route to login users/admin and receive a token to use for authentication and authorization.
- Authentication: None
- Authorization: No Auth

Request Body:

admin:

```
{
    "user_name": "admin_user",
    "email": "admin@email.com",
    "password": "password123",
    "admin": "True"
}
```

user_1:

```
{
    "user_name": "user_1",
    "email": "user1@email.com",
```

```
    "password": "123456",
    "admin": "False"
}
```

user_2:

```
{
    "user_name": "user_2",
    "email": "user2@email.com",
    "password": "123456",
    "admin": "False"
}
```

Request Response:

admin:

```
{
    "user": "admin@email.com",
    "token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTY3OTE5
ODU3NCwianRpIjoiODg2ZmFkYjQtODc5OC00MTZkLWIwMzUtOGI0OGE3MTZhNTYwIiwidHlwZS
I6ImFjY2VzcyIsInN1YiI6IjEiLCJuYmYiOjE2NzkxOTg1NzQsImV4cCI6MTY3OTI4NDk3NH0.
BDVYYZcuoviaL0QbMnmr7yw8M7KEYwkpwMI8Weeo_RU"
}
```

user_1:

```
{
    "user": "user1@email.com",
    "token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTY3OTE5
ODU2MywianRpIjoiNDQ5MDJmZWMtYjYwMS00ZTExLWExY2YtOTM1NTMwYjBhOWM1IiwidHlwZS
I6ImFjY2VzcyIsInN1YiI6IjIiLCJuYmYiOjE2NzkxOTg1NjMsImV4cCI6MTY3OTI4NDk2M30.
VwvzkxtRH17ssTAKlhROHrOboo4_R9IohVhQgF5Ug3o"
}
```

user_2:

```
{
    "user": "user2@email.com",
    "token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTY3OTE5
ODUzMiwianRpIjoiYThkOTg4MTctZjY1NC00YTY4LWE4MDgtNjBlYWI0MzFlMGIyIiwidHlwZS
I6ImFjY2VzcyIsInN1YiI6IjMiLCJuYmYiOjE2NzkxOTg1MzIsImV4cCI6MTY3OTI4NDkzMn0.
```

```
VjuiG9LMQ-CZp2J77SPZfrJe8MPdaKFbZiX_TQqK-HA"
}
```

/auth/register

Method: POST

- Arguments: None
- Description: Registers a new user in the database
- Authentication: None
- Authorization: No Auth

Request Body:

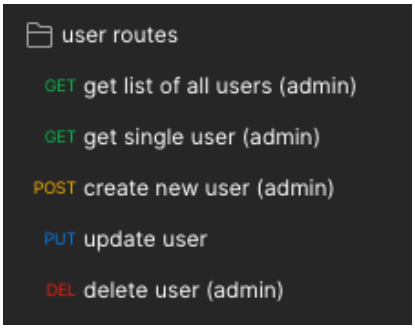
```
{
    "user_name": "user_3",
    "email": "user3@email.com",
    "password": "123456",
    "admin": false
}
```

Request Response:

```
{
    "user": "user3@email.com",
    "token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTY3OTE5
NzgzMSwianRpIjoiNmQ5NTQwNGUtZDdmNS00MTYzLTliMTEtMmM4NTUxY2NjMGYyIiwidHlwZS
I6ImFjY2VzcyIsInN1YiI6IjQiLCJuYmYiOjE2NzkxOTc4MzEsImV4cCI6MTY3OTI4NDIzMX0.
nE7t2EEbJGX9KLppNNzYgF9Fw56xVHextLwdquo33bk"
}
```

User Routes

User Routes in Postman



/users/

Method: GET

- Arguments: None
- Description: A route that returns all users in the database
- Authentication: JWT Required
- Authorization: Bearer Token (admin)

Request Body:

None

Request Response:


```
[
    {
        "id": 1,
        "user_name": "admin_user",
        "email": "admin@email.com",
        "password":
"$2b$12$9wqxeEbEt1tmRRhqQt16.ONIZ38370XzNYE7TsXb7zwU1qUmY6ZBC",
        "admin": true
    },
    {
        "id": 2,
        "user_name": "user_1",
        "email": "user1@email.com",
        "password":
"$2b$12$hCYX/w/k/c6X8YE5J2aAke/P3nEd8LL4ULY4olSSwGgrtoa1TYBZi",
        "admin": false
    },
    {
        "id": 3,
        "user_name": "user_2",
        "email": "user2@email.com",
        "password":
"$2b$12$xo93hpPvyerBd6vBtlNEQuDooE/5/uSfavA0SyfcIJx5hFs5FO0qi",
        "admin": false
    }
]
```


/users/<int:id>/

Method: GET

- Arguments: The user_id (integer) being searched for
- Description: A route that returns a single user in the database
- Authentication: JWT Required

- Authorization: Bearer Token (admin)

Request Body:

None

Request Response:

URL: 127.0.0.1:5000/users/2


```
{
    "id": 2,
    "user_name": "user_1",
    "email": "user1@email.com",
    "password":
"$2b$12$hCYX/w/k/c6X8YE5J2aAke/P3nEd8LL4ULY4olSSwGgrtoa1TYBZi",
    "admin": false
}
```


/users/

Method: POST

- Arguments: None
- Description: A route that allows an admin user to create a new user
- Authentication: JWT Required
- Authorization: Bearer Token (admin)

Request Body:


```
{
    "user_name": "user_4",
    "email": "user4@email.com",
    "password": "123456",
    "admin": false
}
```


Request Response:


```
{
    "id": 4,
    "user_name": "user_4",
    "email": "user4@email.com",
    "password":
"$2b$12$HE6iOE8piNHoFVW8KBp9s.1sMd48Y4RS2TumGnlZVQggm77O4KTmW",
```

```
    "admin": false
}
```

/users/<int:id>/

Method: PUT

- Arguments: The user_id (integer) being searched for
- Description: A route that allows a user to update self. (except admin field)
- Authentication: JWT Required
- Authorization: Bearer Token (user)

Request Body:

URL: 127.0.0.1:5000/users/2

```
{
    "user_name": "user_1_updated",
    "email": "user1_updated@email.com",
    "password": "123456"
}
```

Request Response:

```
{
    "id": 2,
    "user_name": "user_1_updated",
    "email": "user1_updated@email.com",
    "password":
"$2b$12$yeTShfa735LT9h5psbTLRe6zye1rEg4UOYYoffQFAnMptWhjOENvG",
    "admin": false
}
```

/users/<int:id>/

Method: DELETE

- Arguments: The user_id (integer) being searched for
- Description: A route that allows an admin to delete a user.
- Authentication: JWT Required
- Authorization: Bearer Token (admin)

Request Body:

None

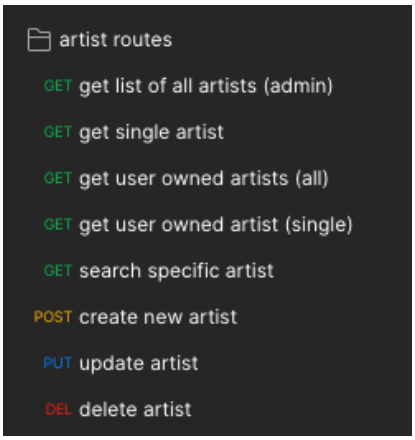
Request Response:

URL: 127.0.0.1:5000/users/3


```
{
    "id": 3,
    "user_name": "user_2",
    "email": "user2@email.com",
    "password":
"$2b$12$eHqFMqqW1PaIq7rMsKc6DewsuezGvDc6mnZYUCXQ0.UgD/YeBqKU2",
    "admin": false
}
```


Artist Routes

Artist Routes in Postman



/artists/

Method: GET

- Arguments: None
- Description: A route that returns all artists in the database
- Authentication: JWT Required
- Authorization: Bearer Token (admin)

Request Body:

None

Request Response:


```
[
    {
        "id": 1,
        "artist_name": "Aphex Twin"
```

```
    },
    {
        "id": 2,
        "artist_name": "Chaos In The CBD"
    },
    {
        "id": 3,
        "artist_name": "Jimmy Whoo"
    }
]
```

/artists/<int:id>/

Method: GET

- Arguments: The artist_id (integer) being searched for
- Description: A route that returns a single artist in the database
- Authentication: JWT Required
- Authorization: Bearer Token (user)

Request Body:

None

Request Response:

URL: 127.0.0.1:5000/artists/1

```
{
    "id": 1,
    "artist_name": "Aphex Twin"
}
```

/artists/user/artists/

Method: GET

- Arguments: None
- Description: A route that returns all artists related to a specifc user in the database
- Authentication: JWT Required
- Authorization: Bearer Token (user)

Request Body:

None

Request Response:

```
[
    {
        "id": 2,
        "artist_name": "Chaos In The CBD"
    },
    {
        "id": 3,
        "artist_name": "Jimmy Whoo"
    }
]
```

/artists/user/<int:id>

Method: GET

- Arguments: The artist_id (integer) being searched for
- Description: A route that returns all artists related to a specifc user in the database
- Authentication: JWT Required
- Authorization: Bearer Token (user)

Request Body:

None

Request Response:

URL: 127.0.0.1:5000/artists/user/1

```
{
    "id": 1,
    "artist_name": "Aphex Twin"
}
```

/artists/search?artist_name=<name_goes_here>

Method: GET

- Arguments: The name of the artist being searched for
- Description: A route that returns a single artist by name
- Authentication: JWT Required
- Authorization: Bearer Token (any user)

Request Body:

None

Request Response:

URL: 127.0.0.1:5000/artists/search?artist_name=Aphex Twin

```
{
    "id": 1,
    "artist_name": "Aphex Twin"
}
```

/artists/

Method: POST

- Arguments: None
- Description: A route that allows any user to create a new artist in the databse
- Authentication: JWT Required
- Authorization: Bearer Token (any user)

Request Body:

```
{
    "artist_name": "new_artist"
}
```

Request Response:

```
{
    "id": 4,
    "artist_name": "new_artist"
}
```

/artists/<int:id>

Method: PUT

- Arguments: The artist_id (integer) being searched for
- Description: A route that allows a user to update an artist that is related to the user in the databse
- Authentication: JWT Required
- Authorization: Bearer Token (user)

Request Body:

```
{
    "artist_name": "Aphex Twin (updated)"
}
```

Request Response:

URL: 127.0.0.1:5000/artists/1

```
{
    "id": 1,
    "artist_name": "Aphex Twin (updated)"
}
```

/artists/<int:id>

Method: PUT

- Arguments: The artist_id (integer) being searched for
- Description: A route that allows a user to delete an artist that is related to the user in the databse
- Authentication: JWT Required
- Authorization: Bearer Token (user)

Request Body:

None

Request Response:

URL: 127.0.0.1:5000/artists/2

```
{
    "id": 2,
    "artist_name": "Chaos In The CBD"
}
```

R7: Third Party Services

The foundation of this RESTful API is built in Flask, a web application framework designed for basic routing, request handling, and response generation. The framework offers a development server and a suite of high-level components that work together to create a powerful and scalable solution. Flask is a lightweight and flexible "micro" framework that enables developers to extend its functionality using various packages, ensuring the API remains fast and adaptable to changing requirements.

The following are the third-party packages that have been used to build this project. The "requirements.txt" file contains a comprehensive list of all the dependencies and requirements needed for the project.

SQLAlchemy

SQLAlchemy is a popular Object-Relational Mapping (ORM) tool used in the development of Python-based applications. It allows developers to work with relational databases in a more Pythonic manner, by representing database tables as Python classes, and rows within those tables as instances of those classes. In the app, SQLAlchemy is used to manage the interactions with the database. It provides an abstraction layer between the Python code and the underlying database, allowing the focus to remain on the business logic of the application, rather than the details of how data is stored and retrieved.

"SQLAlchemy as an ORM means it exposes a model-based API atop database tables so that you don't need to think about the database at all most of the times but to focus on your business logic." (enqueuezero.com, n.d.)

By defining classes that represent the database tables, SQLAlchemy generates the SQL code needed to perform various operations, such as querying the database, inserting new records, and updating existing ones. It also provides support for database migrations, which can help to ensure the database schema stays in sync with changes made to the application over time.

Psycopg2

Psycopg2 is a PostgreSQL database adapter for Python that allows developers to interact with PostgreSQL databases using Python code. It provides a set of Python modules and methods that allow applications to connect to a PostgreSQL database, execute SQL queries, and retrieve results.

In the app, Psycopg2 is used to connect to the PostgreSQL database and perform CRUD (Create, Read, Update, Delete) operations on the database tables. It allows the Python code to interact with the database using SQL commands, such as SELECT, INSERT, UPDATE, and DELETE.

By using Psycopg2, the app can easily communicate with the PostgreSQL database and retrieve data from the various tables, which are Users, Collections, Artists, Records, and Tracks.

Flask-Marshmallow

Flask-Marshmallow is a Flask extension that provides integration between Flask and Marshmallow, a popular Python library for object serialization and deserialization. It simplifies the process of converting Python objects to and from JSON data, which is a common format used in RESTful APIs. In the app, Flask-Marshmallow is used to serialize and deserialize data when communicating with the API endpoints.

"Flask-Marshmallow is a thin integration layer for Flask (a Python web framework) and marshmallow (an object serialization/deserialization library) that adds additional features to marshmallow" (flask-marshmallow.readthedocs.io, n.d.)

It allows the Python code to easily convert the data from the database tables, into JSON format that can be returned by the API endpoints. By defining schema classes that map to the database tables, Flask-Marshmallow generates the JSON data that is returned by the API endpoints. This allows the API to provide a consistent and well-defined data format that can be consumed by other applications and services.

Python-Dotenv

Python-Dotenv is a Python library that allows developers to load environment variables from a .env file in the project directory. This library simplifies the process of managing environment variables by allowing developers to keep sensitive information separate from their code, and to easily switch between different environment configurations.

"When a Python process is created, the available environment variables populate the os.environ object which acts like a Python dictionary." (www.doppler.com, n.d.)

Python-Dotenv is used in conjunction with Flask's config object, which provides a way to manage application-wide configuration variables. The config object can be used to set default values for configuration variables, and to load values from environment variables or configuration files such as .env.

Flask-Bcrypt

Flask-Bcrypt is a package that provides password hashing and verification functionality for Flask applications. It is used to securely store and manage user passwords in the app's database. In the app, Flask-Bcrypt is used to encrypt and store user passwords when they create an account or update their password. When a user logs in, their password is verified by comparing the stored encrypted password with the hashed version of the password they entered during login.

"Password hashing is the process of turning a password into alphanumeric letters using specific algorithms. Some popular algorithms for password hashing include bcrypt and SHA." (Patel, 2022)

Using Flask-Bcrypt to hash and verify passwords adds an extra layer of security to the app, making it more difficult for attackers to gain access to user accounts even if they manage to obtain the password hashes from the database.

Flask-JWT-Extended

Flask-JWT-Extended is a package that provides JSON Web Token (JWT) functionality for Flask applications. It is used to provide authentication and access control to the API endpoints. In the app, Flask-JWT-Extended is used to generate JWTs when users are successfully authenticated and include the token in subsequent requests to protected endpoints.

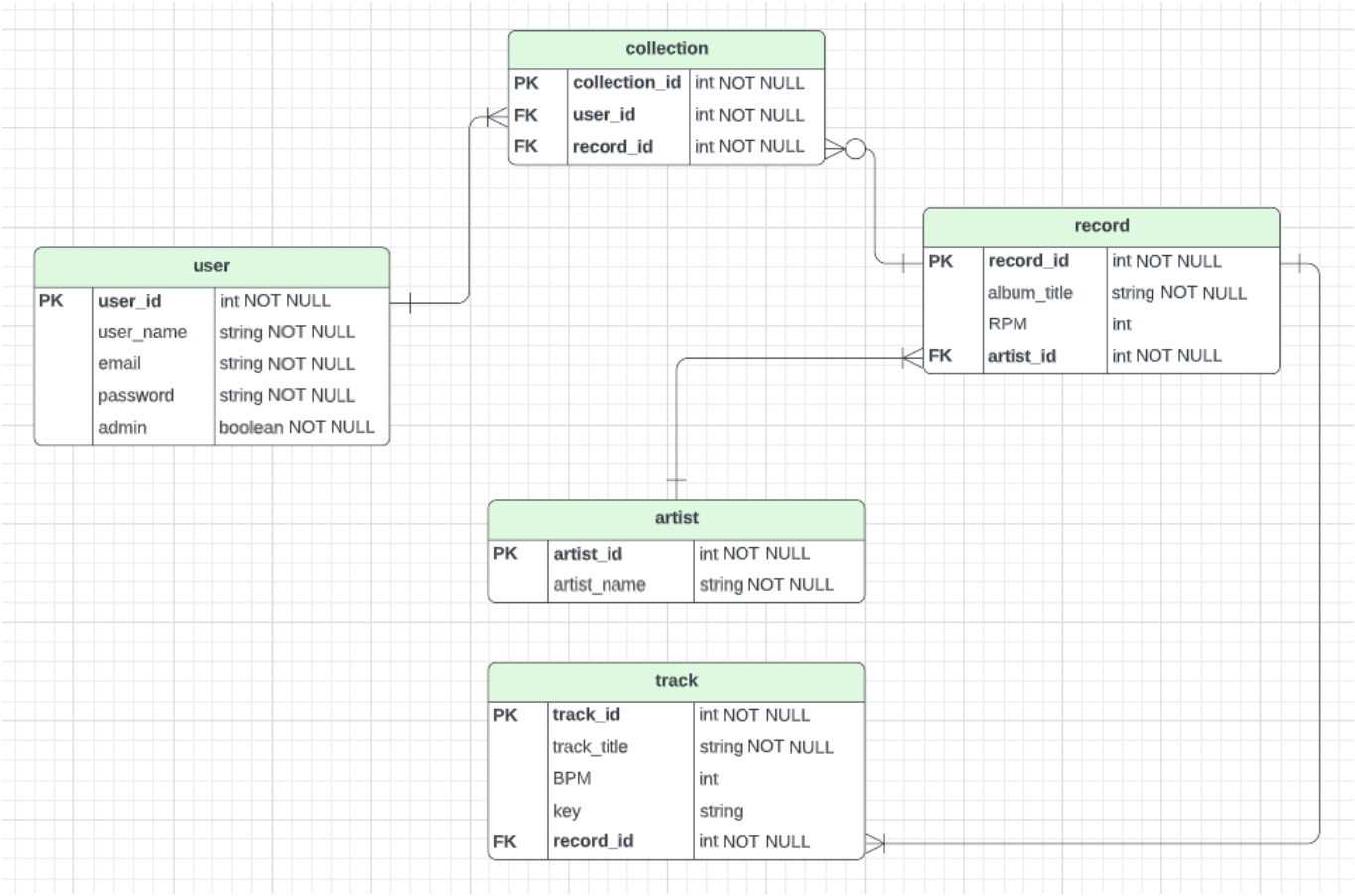
"To access a jwt_required protected view you need to send in the JWT with each request. By default, this is done with an authorization header." (flask-jwt-extended.readthedocs.io, n.d.)

The JWT contains encoded information about the user's identity and permissions. When a user makes a request to a protected endpoint, Flask-JWT-Extended checks the JWT for validity and authenticity. If the token is valid and has not expired, the user is granted access to the endpoint. Flask-JWT-Extended can also be used to control access to specific endpoints based on the user's permissions.

R6 / R9: Explanation of ERD and Database Relations Implementation

Presented below is an Entity Relationship Diagram (ERD) that illustrates the connections between the tables within the relational database implemented in this project. Each table is depicted as an entity in the diagram, with the attributes or fields of the database table being represented as corresponding elements in the ERD. The primary key, which serves as the unique identifier for each table, is included in every entity. The third column of each entity in the ERD shows the attribute's datatype and indicates if it is mandatory for the database table entry (NOT NULL). Additionally, foreign keys are present in some tables, signifying a connection between two tables. This relationship is also shown through the crow's foot notation utilized in the diagram.

ERD



Entities

User

The “user” table retains details regarding users who intend to store information about their vinyl record collection in the database. The attributes contained in the table are user_id (allowing them to access the system) user_name (a self-selected username for the application), email (utilized for user identification), password (used for authentication purposes), and admin, which stores a boolean value that authenticates and authorizes users. The relationship between the user and record table is many to many, as many users can own many records and many records can belong to many users. Because of this a new entity titled “collection” was created to represent this relationship between user and records.

Collection

Since the many to many relationship between user and record was handled by the creation of this table, we now have a 1 to many relationship between user and collection. The attributes included in this entity are collection_id, user_id and record_id. With the former being the primary key and the others, foreign keys. The purpose of this table, is to create a unique identifier (PK) that points to each collection in the database (relationship between user and record). It's not a table that needs to be accessed by regular users, but rather a "background" table doing the job of handling a many to many relationship in the database.

Record / Artist

The record table acts as a kind of centre point to the application. Because it is linked to users through collections and has a dependent table (tracks). The PK in this table is record_id, which is a unique identifier for every album (record) entered into the database. The attributes of this table include album_title (name of the album), RPM (rotations per minute eg: 30 or 45) and artist_id (FK), which is the PK from the artist table.

Originally, the artist_id was included as an attribute in this table but was later moved out to create a table of its own. This was done to help normalise the database as you might find the same artist many times in this table. We also now have a 1 to many relationship between the artist and record tables because 1 artist can have many records, but each record belongs to a single artist. The artist table was kept simple and only includes the artist_id and artist_name as it was moved out for the purpose of normalization. Since we have a 1 to many relationship here, we can think of the record table as being dependent on the artist table. In order for a record to exist, there must be an artist.

Track

The track table lies at the end of the chain so to speak. Because no tables depend on the track table, if a track was to be deleted from the database, no other data should be affected. The track table includes a track_id (identified each unique track), track_title (name of the track), BPM (beats per minute or tempo eg: 120bpm), key (eg: A Minor), and record_id, which represents the 1 to many relationship between the record and track tables. This relationship is one to many because 1 record can have many tracks but each track belongs to only 1 record.

As you can see from the ERD, not every attribute was made nullable. The ones that are nullable, are attributes that without an entry, would not effect the relationship between the entities in the database, helping normalize and strengthen the database structure.

Relationships again:

- Many to Many between the user and record tables: creating a collection table.
- 1 to Many between the artist and record tables.
- 1 to Many between the record and track tables.

R8: Project Models and their Relationships with each other

The models in this project are implemented using SQLAlchemy ORM for handling database operations. There are five main classes in the code representing the tables in the database: User, Collection, Record, Artist, and Track. Each class inherits from db.Model, which is a base class for all models in SQLAlchemy.

User class: Represents a user in the system

- **__ tablename __:** Sets the name of the table in the database to "users". id: Primary key column with auto-incrementing integer values.
- **user_name, email, password, and admin:** Additional attributes with various constraints (e.g., nullable=False means the attribute must have a value, and unique=True means the attribute value must be unique across all users).
- **collections:** One-to-many relationship between User and Collection. When a user is deleted, all associated collections will also be deleted due to the "cascade" parameter.

Collection class: Represents a collection of records owned by a user

- **__ tablename __:** Sets the table name to "collections".
- **id:** Primary key column.
- **user_id:** Foreign key column referencing the 'users.id' column, establishing a relationship with the User table.
- **record_id:** Foreign key column referencing the 'records.id' column, establishing a relationship with the Record table.

Record class: Represents a vinyl record (album)

- **__ tablename __:** Sets the table name to "records".
- **id:** Primary key column.
- **album_title, rpm, and artist_id:** Additional attributes, with 'artist_id' being a foreign key referencing the 'artists.id' column.
- **collections:** One-to-many relationship with Collection. When a record is deleted, all associated collections will also be deleted due to the "cascade" parameter.
- **tracks:** One-to-many relationship with Track. When a record is deleted, all associated tracks will also be deleted due to the "cascade" parameter.

Artist class: Represents a music artist

- **__ tablename __:** Sets the table name to "artists".
- **id:** Primary key column.
- **artist_name:** Additional attribute.
- **records:** One-to-many relationship with Record. When an artist is deleted, all associated records will also be deleted due to the "cascade" parameter.

Track class: Represents a track in a record (album)

- **__ tablename __:** Sets the table name to "tracks".
- **id:** Primary key column.
- **track_title, bpm, key, and record_id:** Additional attributes, with 'record_id' being a foreign key referencing the 'records.id' column.

Associations between the models:

- A User can have multiple Collections.
- A Collection is associated with one User and one Record.
- An Artist can have multiple Records.
- A Record is associated with one Artist and can have multiple Tracks and Collections.
- A Track is associated with one Record.

The code uses SQLAlchemy's relationship() function to establish these associations. The backref parameter creates a reverse relationship, making it easy to navigate from one side of the relationship to the other (e.g. from Record to Artist, and vice versa). The cascade parameter ensures that when a parent record is deleted, all related child records are deleted as well.

To elborate further on the code:

User and Collection:

```
collections = db.relationship(
    "Collection",
    backref="user",
    cascade="all, delete"
)
```

This line establishes that one user can have multiple collections. The backref parameter creates a reverse relationship, allowing you to access the User model from the Collection model using the "user" attribute.

In the Collection class, there are foreign key columns for both user_id and record_id:

```
user_id = db.Column(db.Integer, db.ForeignKey('users.id'), nullable=False)
record_id = db.Column(db.Integer, db.ForeignKey('records.id'),
nullable=False)
These columns reference the primary keys of the User and Record tables,
respectively, to create the associations between these models.
```

Artist and Record:

```
records = db.relationship(
    "Record",
    backref="artist",
    cascade="all, delete"
)
```

This line establishes that one artist can have multiple records. The backref parameter creates a reverse relationship, allowing you to access the Artist model from the Record model using the "artist" attribute.

In the Record class, the artist_id column is a foreign key referencing the 'artists.id' column:

```
artist_id = db.Column(db.Integer, db.ForeignKey('artists.id'),
nullable=False)
This column creates the association between the Record and Artist models.
```

Record and Track:

```
tracks = db.relationship(
    "Track",
    backref="record",
    cascade="all, delete"
)
```

This line establishes that one record can have multiple tracks. The backref parameter creates a reverse relationship, allowing you to access the Record model from the Track model using the "record" attribute.

In the Track class, the record_id column is a foreign key referencing the 'records.id' column:

```
record_id = db.Column(db.Integer, db.ForeignKey('records.id'),
nullable=False)
```

This column creates the association between the Track and Record models.

Collection and Record:

As mentioned earlier, the Collection class has a foreign key column record_id, which references the primary key of the Record table:

```
record_id = db.Column(db.Integer, db.ForeignKey('records.id'),
nullable=False)
```

```
collections = db.relationship(
    "Collection",
    backref="record",
    cascade="all, delete"
)
```

This line establishes that one record can be part of multiple collections. The backref parameter creates a reverse relationship, allowing you to access the Record model from the Collection model using the "record" attribute.

R10: Planning and Tracking of Tasks

Trello Board: T2A2 - Vinyl Data API

Trello was chosen as the tool to plan and track tasks for this project. This allowed for small portions of work to be defined as checklists inside cards that represent the larger concept.

The Trello workspace was separated into 3 main parts:

To Do: An organized schedule of tasks that have been arranged in order of importance and are prepared for implementation.

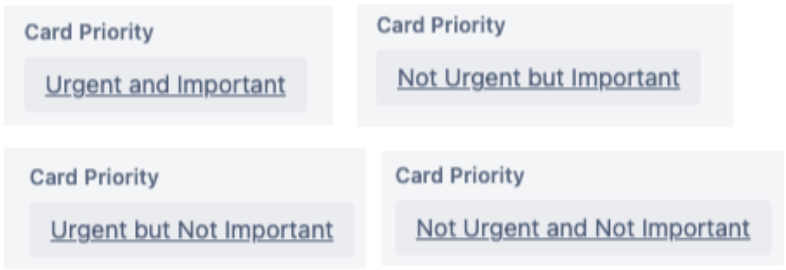
Doing: Tasks that are currently being worked on. This is the section that was revisited the most, especially to view the questions/reminders card. This card was extremely useful as a one-stop place for any ideas that popped up during production. Questions that needed answers, reminders or instructions on how to do something later.

Done: When cards were completed, they would be sent to this list.

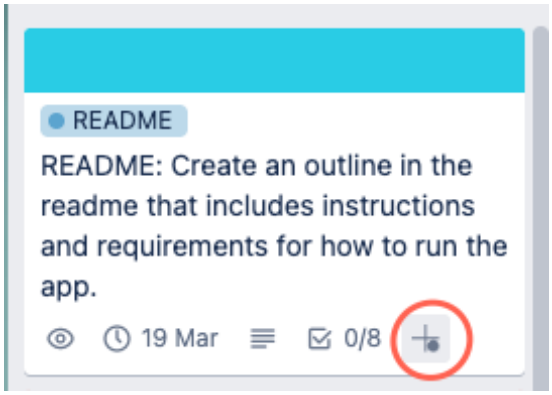
Each card was given a cover with a certain colour to represent the kind of work that needed to be done. This was used as an eye-catcher and helped to quickly identify what “type” of work was left to do. Red represented rubric questions, yellow represented the core modules of the code in terms of MVC, purple was for authorization and authentication, dark green was for questions/ reminders and blue was generally for anything to do with the readme.md file. This exact pattern had exceptions, eg: when the Trello board was initially created, there were some blue and green cards made for tasks to do with starting the project and working on the ERD. Because they were done so soon, they were left the way they were. Colour coding was also added as labels inside the cards.

When a card is opened there are certain attributes that helped define how urgent or important a task was. A power-up was installed called Card Priority Badge. The power-up meant that you could assign each card a level of importance (essentially from 1-4) which included an icon that would be shown on the outside of the card so they could be easily seen when scanning through the Trello board.

Card Priority Badges looked like this inside the card:

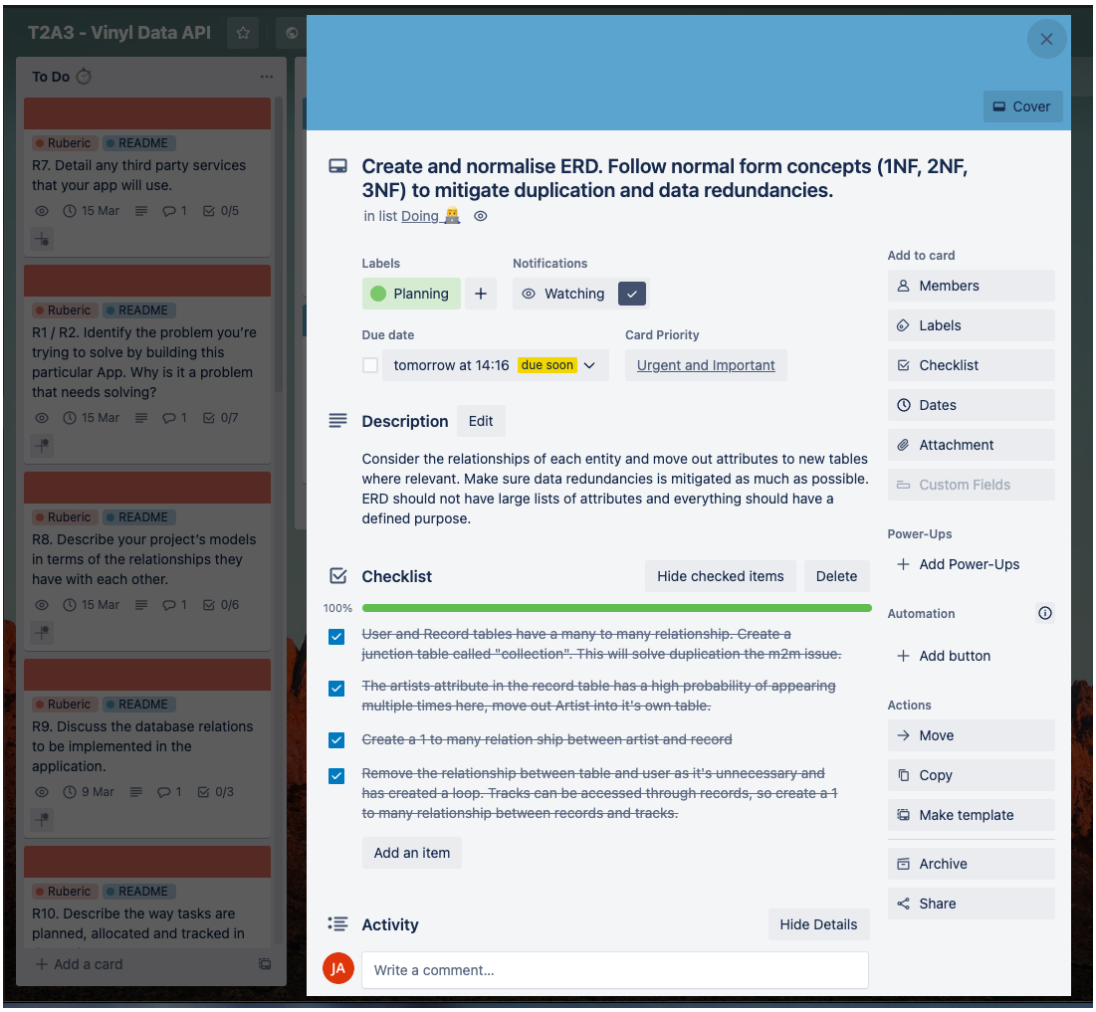


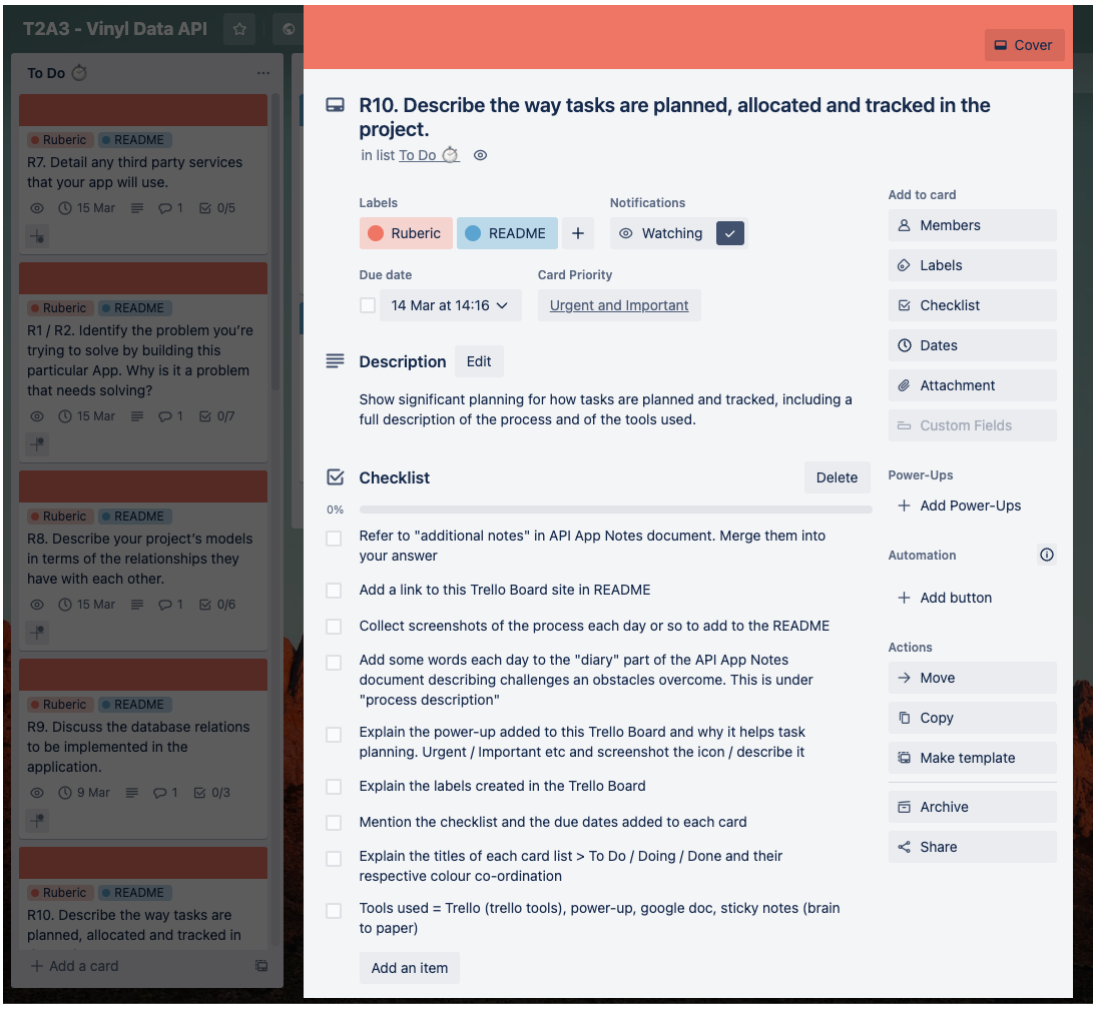
Card Priority Badges looked like this outside the card:

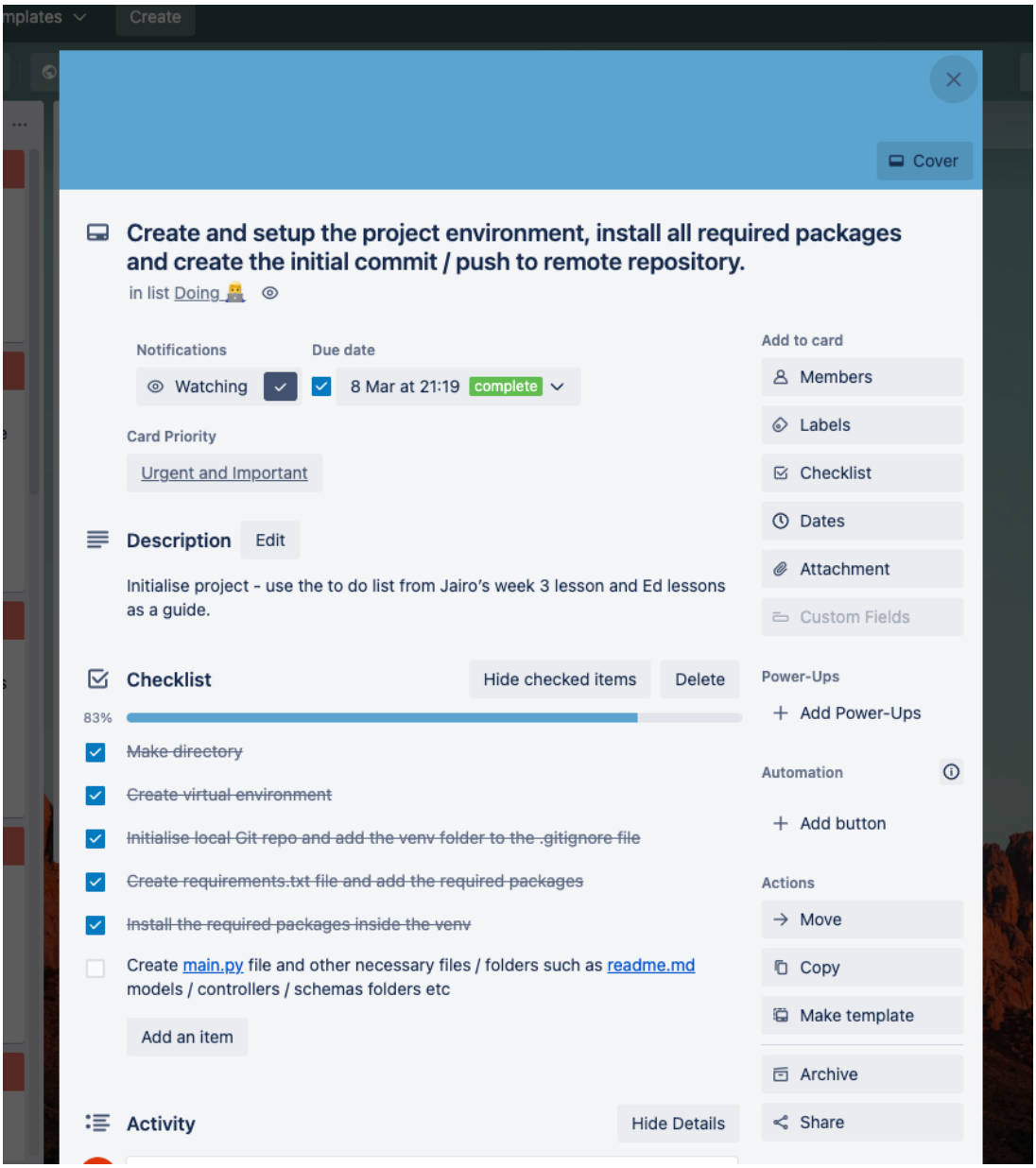


A due date reminder was also set to each card that was designed to that 1 day before each card was due, I would receive a notification via email to prompt me to view the card and read over the checklist. As the tasks were completed, the checklist was gradually ticked off until the card was complete.

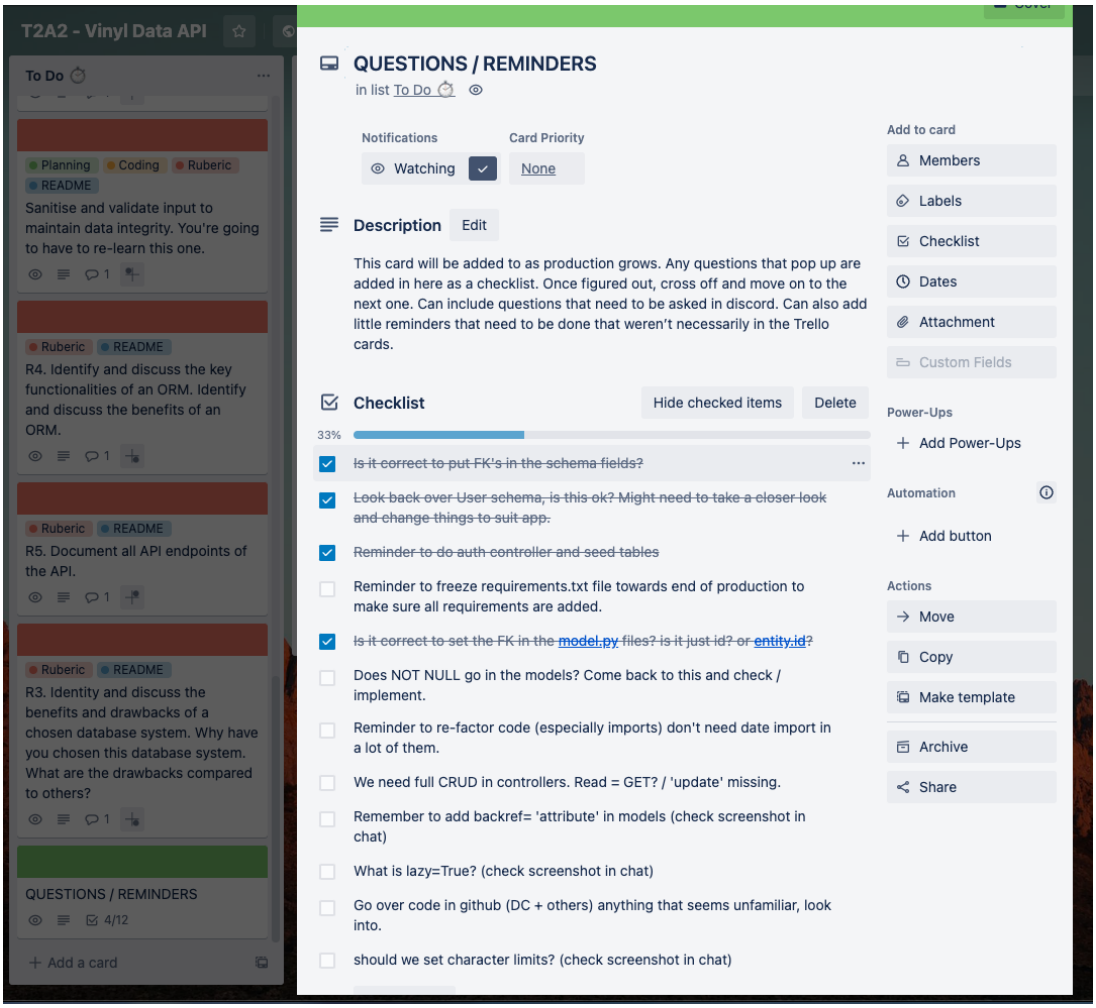
Below are screenshots capturing the gradual completion of tasks within the cards:

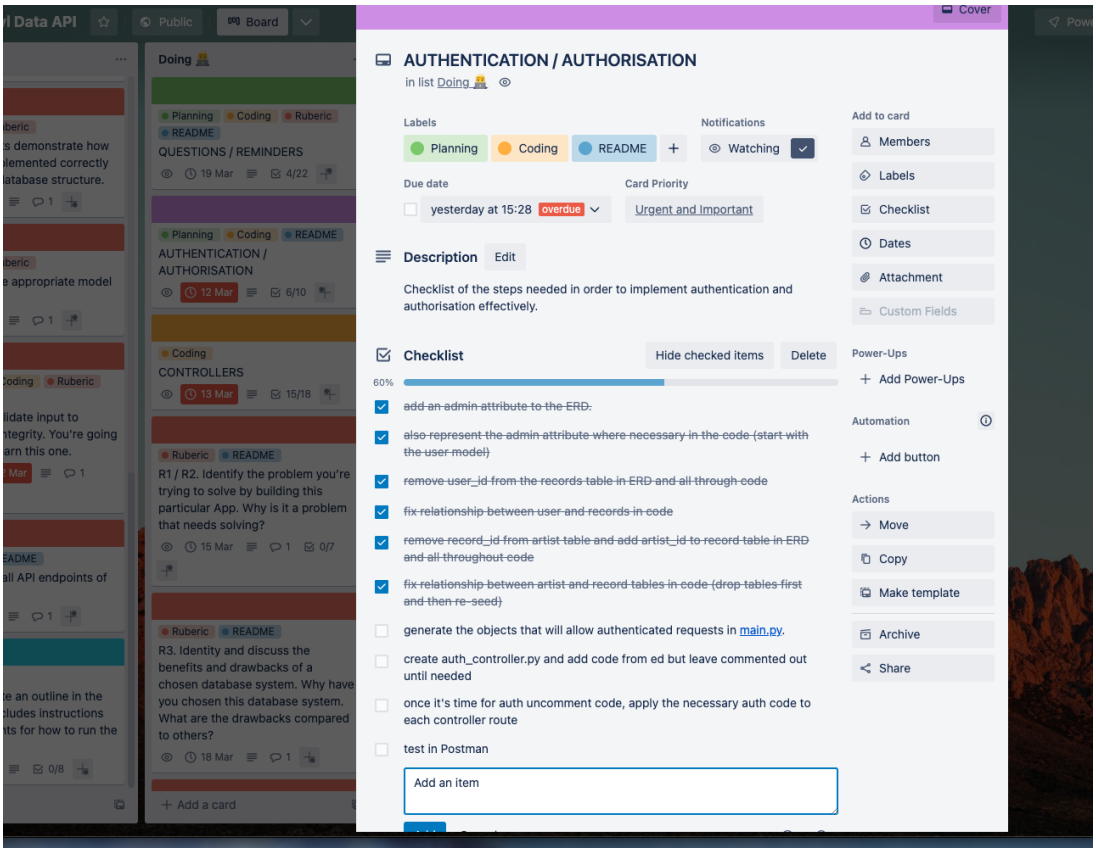

T2A3 - Vinyl Data API
To Do
Ruberic README
R7. Detail any third party services that your app will use.
15 Mar 1 0/5
Ruberic README
R1 / R2. Identify the problem you're trying to solve by building this particular App. Why is it a problem that needs solving?
15 Mar 1 0/7
Ruberic README
R8. Describe your project's models in terms of the relationships they have with each other.
15 Mar 1 0/6
Ruberic README
R9. Discuss the database relations to be implemented in the application.
9 Mar 1 0/3
Ruberic README
R10. Describe the way tasks are planned, allocated and tracked in
Add a card
Cover
Create and normalise ERD. Follow normal form concepts (1NF, 2NF, 3NF) to mitigate duplication and data redundancies.
in list Doing
Labels
Planning
Notifications
Watching
Due date
tomorrow at 14:16 due soon
Card Priority
Urgent and Important
Description
Edit
Consider the relationships of each entity and move out attributes to new tables where relevant. Make sure data redundancies is mitigated as much as possible. ERD should not have large lists of attributes and everything should have a defined purpose.
Checklist
Hide checked items
Delete
100%
User and Record tables have a many to many relationship. Create a junction table called "collection". This will solve duplication the m2m issue.
The artists attribute in the record table has a high probability of appearing multiple times here, move out Artist into it's own table.
Create a 1 to many relation ship between artist and record
Remove the relationship between table and user as it's unnecessary and has created a loop. Tracks can be accessed through records, so create a 1 to many relationship between records and tracks.
Add an item
Activity
Hide Details
JA
Write a comment...
Add to card
Members
Labels
Checklist
Dates
Attachment
Custom Fields
Power-Ups
Add Power-Ups
Automation
Add button
Actions
Move
Copy
Make template
Archive
Share


T2A3 - Vinyl Data API
To Do
Ruberic README
R7. Detail any third party services that your app will use.
15 Mar 1 0/5
Ruberic README
R1 / R2. Identify the problem you're trying to solve by building this particular App. Why is it a problem that needs solving?
15 Mar 1 0/7
Ruberic README
R8. Describe your project's models in terms of the relationships they have with each other.
15 Mar 1 0/6
Ruberic README
R9. Discuss the database relations to be implemented in the application.
9 Mar 1 0/3
Ruberic README
R10. Describe the way tasks are planned, allocated and tracked in
Add a card
Cover
R10. Describe the way tasks are planned, allocated and tracked in the project.
in list To Do
Labels
Ruberic
README
Notifications
Watching
Due date
14 Mar at 14:16
Card Priority
Urgent and Important
Description
Edit
Show significant planning for how tasks are planned and tracked, including a full description of the process and of the tools used.
Checklist
Delete
0%
Refer to "additional notes" in API App Notes document. Merge them into your answer
Add a link to this Trello Board site in README
Collect screenshots of the process each day or so to add to the README
Add some words each day to the "diary" part of the API App Notes document describing challenges an obstacles overcome. This is under "process description"
Explain the power-up added to this Trello Board and why it helps task planning. Urgent / Important etc and screenshot the icon / describe it
Explain the labels created in the Trello Board
Mention the checklist and the due dates added to each card
Explain the titles of each card list > To Do / Doing / Done and their respective colour co-ordination
Tools used = Trello (trello tools), power-up, google doc, sticky notes (brain to paper)
Add an item
Add to card
Members
Labels
Checklist
Dates
Attachment
Custom Fields
Power-Ups
Add Power-Ups
Automation
Add button
Actions
Move
Copy
Make template
Archive
Share

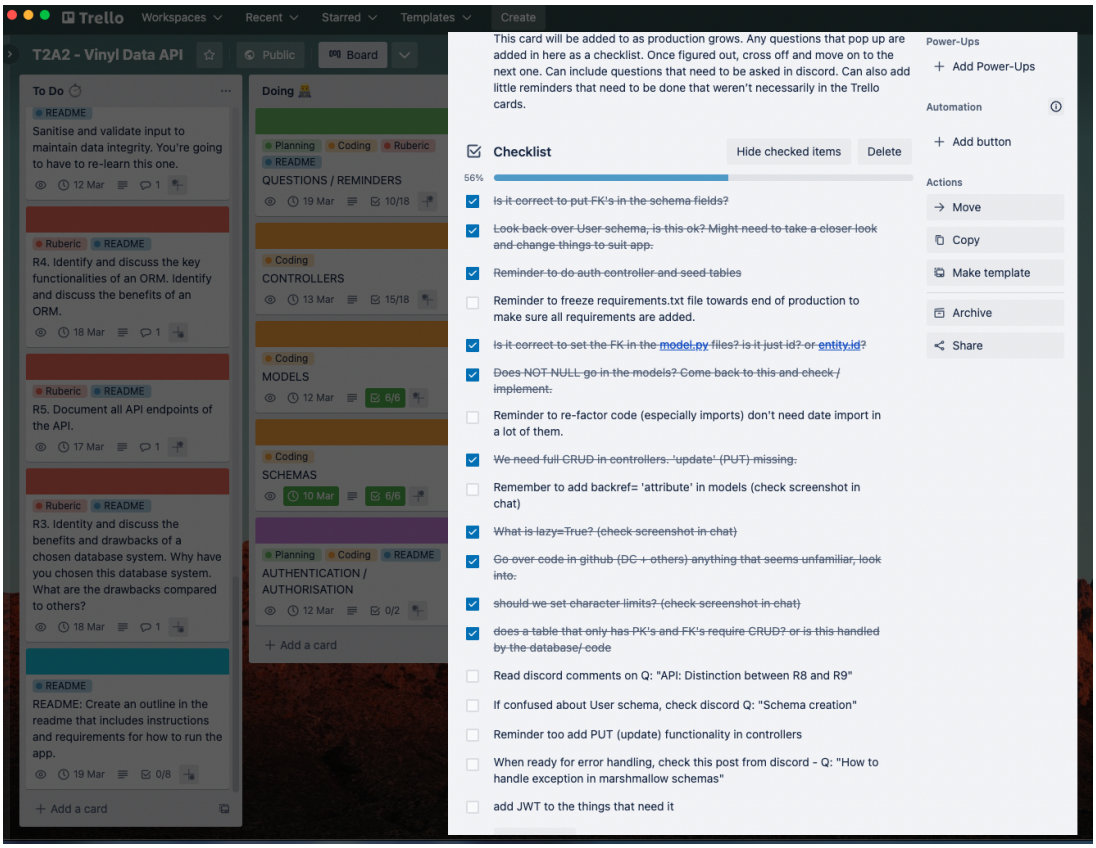

Create
Cover
Create and setup the project environment, install all required packages and create the initial commit / push to remote repository.
in list Doing
Notifications
Watching
Due date
8 Mar at 21:19
complete
Card Priority
Urgent and Important
Description
Edit
Initialise project - use the to do list from Jairo's week 3 lesson and Ed lessons as a guide.
Checklist
Hide checked items
Delete
83%
Make directory
Create virtual environment
Initialise local Git repo and add the venv folder to the .gitignore file
Create requirements.txt file and add the required packages
Install the required packages inside the venv
Create main.py file and other necessary files / folders such as readme.md models / controllers / schemas folders etc
Add an item
Activity
Hide Details
Add to card
Members
Labels
Checklist
Dates
Attachment
Custom Fields
Power-Ups
Add Power-Ups
Automation
Add button
Actions
Move
Copy
Make template
Archive
Share

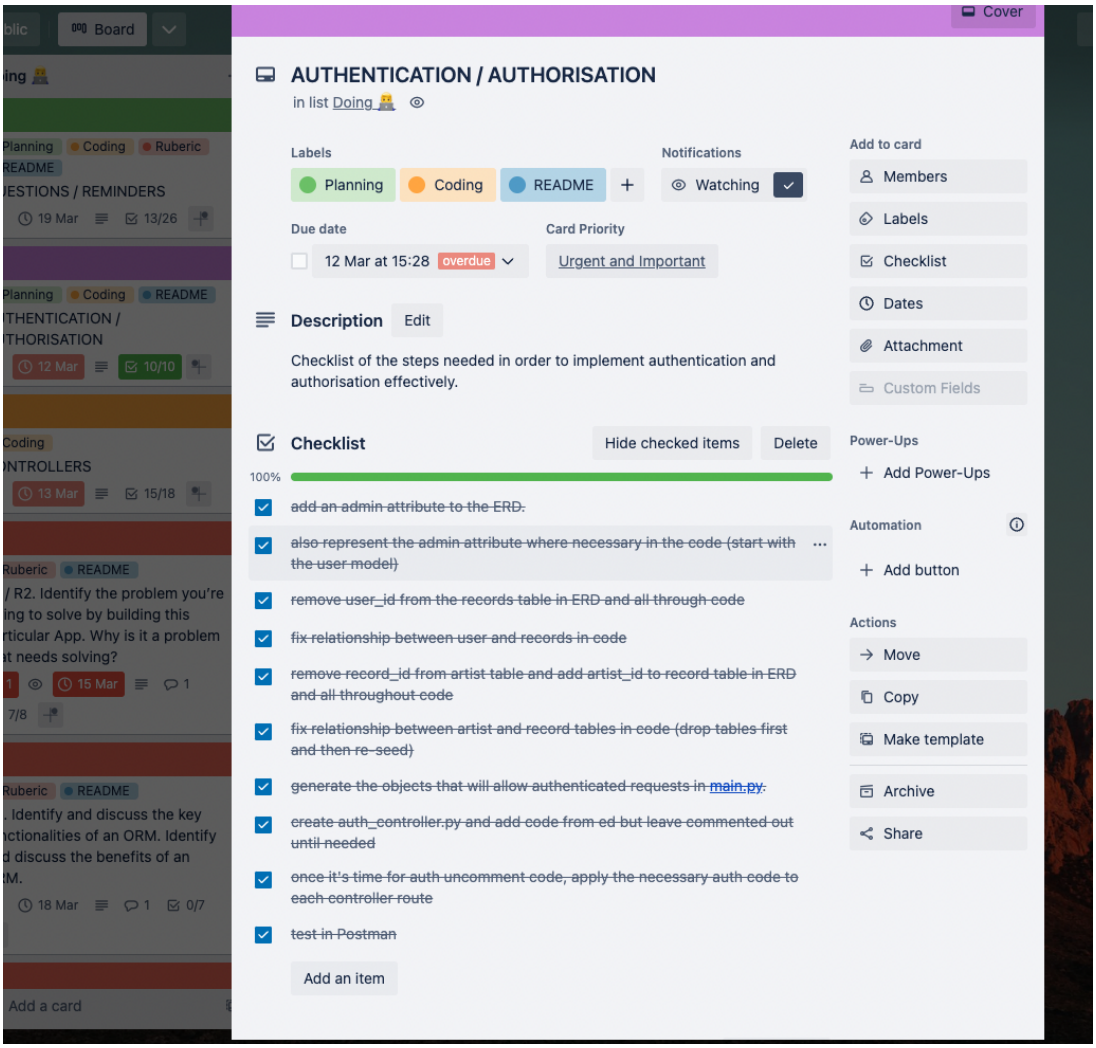

T2A2 - Vinyl Data API
Public
Board
To Do
Ruberic
README
R4. Identify and discuss the key functionalities of an ORM. Identify and discuss the benefits of an ORM.
18 Mar
1
Coding
CONTROLLERS
13 Mar
15/18
Ruberic
README
R5. Document all API endpoints of the API.
17 Mar
1
Ruberic
README
R3. Identity and discuss the benefits and drawbacks of a chosen database system. Why have you chosen this database system. What are the drawbacks compared to others?
18 Mar
1
QUESTIONS / REMINDERS
19 Mar
1/12
Coding
SCHEMAS
10 Mar
0/6
Add a card
Doing
Planning
Create and normalise ERD. Follow normal form concepts (1NF, 2NF, 3NF) to mitigate duplication and data redundancies.
5 Mar
1
4/4
Planning
Build some mock tables in Google Sheets and pre-fill the fields with sample data.
4 Mar
1
3/3
Coding
MODELS
12 Mar
6/6
Add a card
Done
Planning
Get idea approved by educators on Discord. Continue to re-work ERD until it it normalised and all relationships are accounted for. Focus on 1NF 2NF 3NF.
1 Mar
1
4/4
Create and setup the project environment, install all required packages and create the initial commit / push to remote repository.
8 Mar
6/6
Add a card

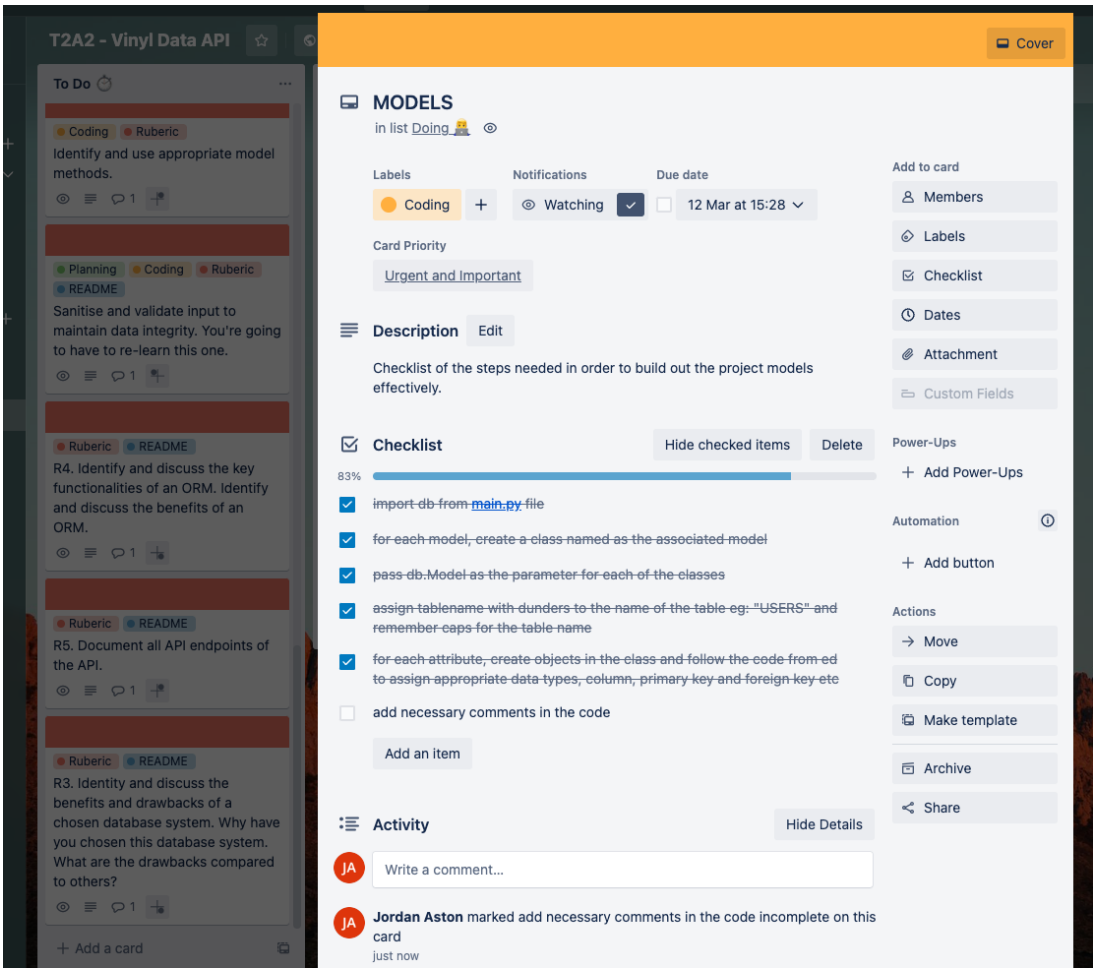

T2A2 - Vinyl Data API
To Do
Planning Coding Ruberic README
Sanitise and validate input to maintain data integrity. You're going to have to re-learn this one.
Ruberic README
R4. Identify and discuss the key functionalities of an ORM. Identify and discuss the benefits of an ORM.
Ruberic README
R5. Document all API endpoints of the API.
Ruberic README
R3. Identity and discuss the benefits and drawbacks of a chosen database system. Why have you chosen this database system. What are the drawbacks compared to others?
QUESTIONS / REMINDERS
4/12
Add a card
QUESTIONS / REMINDERS
in list To Do
Notifications
Watching
Card Priority
None
Description
Edit
This card will be added to as production grows. Any questions that pop up are added in here as a checklist. Once figured out, cross off and move on to the next one. Can include questions that need to be asked in discord. Can also add little reminders that need to be done that weren't necessarily in the Trello cards.
Checklist
Hide checked items
Delete
33%
Is it correct to put FK's in the schema fields?
Look back over User schema, is this ok? Might need to take a closer look and change things to suit app.
Reminder to do auth controller and seed tables
Reminder to freeze requirements.txt file towards end of production to make sure all requirements are added.
Is it correct to set the FK in the model.py files? is it just id? or entity.id?
Does NOT NULL go in the models? Come back to this and check / implement.
Reminder to re-factor code (especially imports) don't need date import in a lot of them.
We need full CRUD in controllers. Read = GET? / 'update' missing.
Remember to add backref= 'attribute' in models (check screenshot in chat)
What is lazy=True? (check screenshot in chat)
Go over code in github (DC + others) anything that seems unfamiliar, look into.
should we set character limits? (check screenshot in chat)
Add to card
Members
Labels
Checklist
Dates
Attachment
Custom Fields
Power-Ups
Add Power-Ups
Automation
Add button
Actions
Move
Copy
Make template
Archive
Share

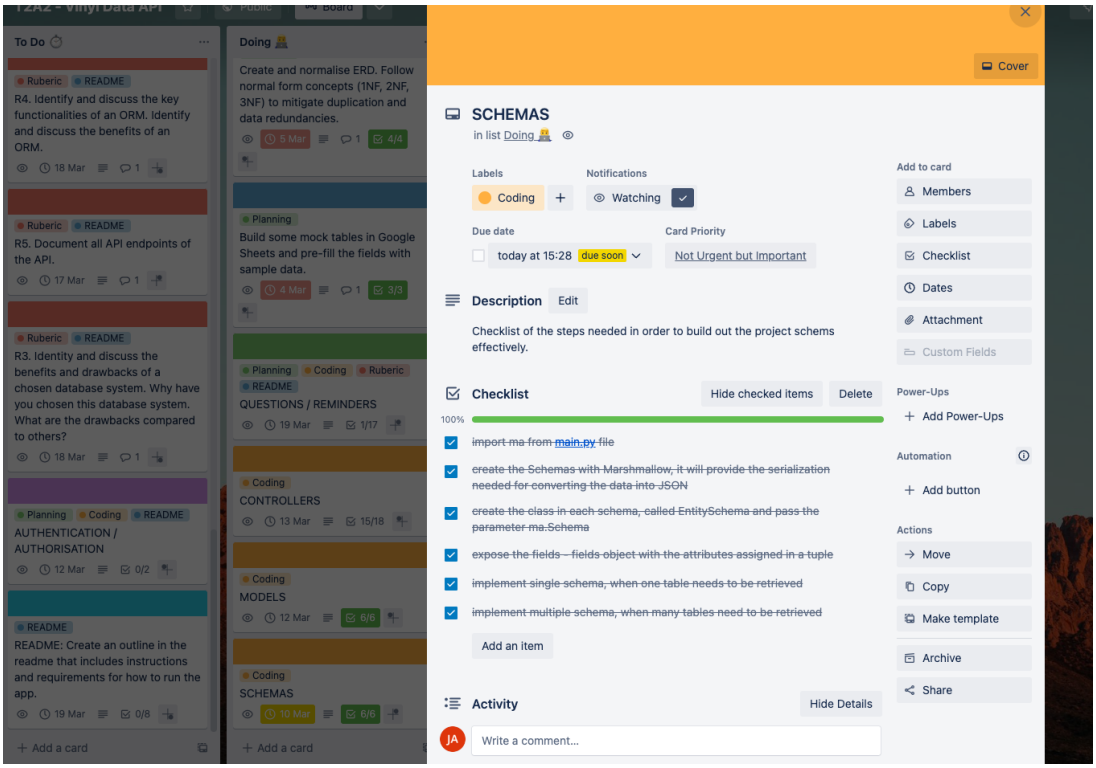

Public
Board
Doing
Planning Coding Ruberic README
QUESTIONS / REMINDERS
19 Mar 4/22
Planning Coding README
AUTHENTICATION / AUTHORISATION
12 Mar 6/10
Coding
CONTROLLERS
13 Mar 15/18
Ruberic README
R1 / R2. Identify the problem you're trying to solve by building this particular App. Why is it a problem that needs solving?
15 Mar 0/7
Ruberic README
R3. Identity and discuss the benefits and drawbacks of a chosen database system. Why have you chosen this database system. What are the drawbacks compared to others?
18 Mar
Add a card
Cover
AUTHENTICATION / AUTHORISATION
in list Doing
Labels
Planning Coding README
Notifications
Watching
Due date
yesterday at 15:28 overdue
Card Priority
Urgent and Important
Description
Edit
Checklist of the steps needed in order to implement authentication and authorisation effectively.
Checklist
Hide checked items
Delete
60%
add an admin attribute to the ERD.
also represent the admin attribute where necessary in the code (start with the user model)
remove user_id from the records table in ERD and all through code
fix relationship between user and records in code
remove record_id from artist table and add artist_id to record table in ERD and all throughout code
fix relationship between artist and record tables in code (drop tables first and then re-seed)
generate the objects that will allow authenticated requests in main.py.
create auth_controller.py and add code from ed but leave commented out until needed
once it's time for auth uncomment code, apply the necessary auth code to each controller route
test in Postman
Add an item
Add to card
Members
Labels
Checklist
Dates
Attachment
Custom Fields
Power-Ups
Add Power-Ups
Automation
Add button
Actions
Move
Copy
Make template
Archive
Share


Trello
Workspaces
Recent
Starred
Templates
Create
T2A2 - Vinyl Data API
Public
Board
To Do
README
README: Create an outline in the readme that includes instructions and requirements for how to run the app.
19 Mar
0/8
Planning
Coding
Ruberic
README
Sanitise and validate input to maintain data integrity. You're going to have to re-learn this one.
1
12 Mar
1
Ruberic
README
R5. Document all API endpoints of the API.
17 Mar
1
Ruberic
README
R8. Describe your project's models in terms of the relationships they have with each other.
1
15 Mar
1
0/6
Ruberic
README
R9. Discuss the database relations to be implemented in the application.
Add a card
Doing
Planning
Coding
Ruberic
README
QUESTIONS / REMINDERS
19 Mar
13/26
Ruberic
README
R10. Describe the way tasks are planned, allocated and tracked in the project.
14 Mar
1
0/9
Add a card
Done
Coding
MODELS
12 Mar
6/6
Coding
SCHEMAS
10 Mar
6/6
Coding
CONTROLLERS
13 Mar
18/18
Planning
Coding
README
AUTHENTICATION / AUTHORISATION
12 Mar
10/10
Planning
Build some mock tables in Google Sheets and pre-fill the fields with sample data.
4 Mar
1
3/3
Planning
Create and normalise ERD. Follow normal form concepts (1NF, 2NF, 3NF) to mitigate duplication and data redundancies.
Add a card


Trello
Workspaces
Recent
Starred
Templates
Create
T2A2 - Vinyl Data API
Public
Board
To Do
README
Sanitise and validate input to maintain data integrity. You're going to have to re-learn this one.
12 Mar
Ruberic README
R4. Identify and discuss the key functionalities of an ORM. Identify and discuss the benefits of an ORM.
18 Mar
R5. Document all API endpoints of the API.
17 Mar
R3. Identity and discuss the benefits and drawbacks of a chosen database system. Why have you chosen this database system. What are the drawbacks compared to others?
18 Mar
README: Create an outline in the readme that includes instructions and requirements for how to run the app.
19 Mar
0/8
Add a card
Doing
Planning Coding Ruberic README
QUESTIONS / REMINDERS
19 Mar
10/18
Coding
CONTROLLERS
13 Mar
15/18
MODELS
12 Mar
6/6
SCHEMAS
10 Mar
AUTHENTICATION / AUTHORISATION
0/2
This card will be added to as production grows. Any questions that pop up are added in here as a checklist. Once figured out, cross off and move on to the next one. Can include questions that need to be asked in discord. Can also add little reminders that need to be done that weren't necessarily in the Trello cards.
Checklist
Hide checked items
Delete
56%
Is it correct to put FK's in the schema fields?
Look back over User schema, is this ok? Might need to take a closer look and change things to suit app.
Reminder to do auth controller and seed tables
Reminder to freeze requirements.txt file towards end of production to make sure all requirements are added.
Is it correct to set the FK in the model.py files? is it just id? or entity.id?
Does NOT NULL go in the models? Come back to this and check / implement.
Reminder to re-factor code (especially imports) don't need date import in a lot of them.
We need full CRUD in controllers. 'update' (PUT) missing.
Remember to add backref= 'attribute' in models (check screenshot in chat)
What is lazy=True? (check screenshot in chat)
Go over code in github (DC + others) anything that seems unfamiliar, look into.
should we set character limits? (check screenshot in chat)
does a table that only has PK's and FK's require CRUD? or is this handled by the database/ code
Read discord comments on Q: "API: Distinction between R8 and R9"
If confused about User schema, check discord Q: "Schema creation"
Reminder too add PUT (update) functionality in controllers
When ready for error handling, check this post from discord - Q: "How to handle exception in marshmallow schemas"
add JWT to the things that need it
Power-Ups
Add Power-Ups
Automation
Add button
Actions
Move
Copy
Make template
Archive
Share


Board
Cover
AUTHENTICATION / AUTHORISATION
in list Doing
Labels
Planning
Coding
README
Notifications
Watching
Due date
12 Mar at 15:28
overdue
Card Priority
Urgent and Important
Description
Edit
Checklist of the steps needed in order to implement authentication and authorisation effectively.
Checklist
Hide checked items
Delete
100%
add an admin attribute to the ERD.
also represent the admin attribute where necessary in the code (start with the user model)
remove user_id from the records table in ERD and all through code
fix relationship between user and records in code
remove record_id from artist table and add artist_id to record table in ERD and all throughout code
fix relationship between artist and record tables in code (drop tables first and then re-seed)
generate the objects that will allow authenticated requests in main.py.
create auth_controller.py and add code from ed but leave commented out until needed
once it's time for auth uncomment code, apply the necessary auth code to each controller route
test in Postman
Add an item
Add to card
Members
Labels
Checklist
Dates
Attachment
Custom Fields
Power-Ups
Add Power-Ups
Automation
Add button
Actions
Move
Copy
Make template
Archive
Share


T2A2 - Vinyl Data API
To Do
Coding Ruberic
Identify and use appropriate model methods.
Planning Coding Ruberic README
Sanitise and validate input to maintain data integrity. You're going to have to re-learn this one.
Ruberic README
R4. Identify and discuss the key functionalities of an ORM. Identify and discuss the benefits of an ORM.
Ruberic README
R5. Document all API endpoints of the API.
Ruberic README
R3. Identity and discuss the benefits and drawbacks of a chosen database system. Why have you chosen this database system. What are the drawbacks compared to others?
Add a card
Cover
MODELS
in list Doing
Labels
Coding
Notifications
Watching
Due date
12 Mar at 15:28
Card Priority
Urgent and Important
Description
Edit
Checklist of the steps needed in order to build out the project models effectively.
Checklist
Hide checked items
Delete
83%
import db from main.py file
for each model, create a class named as the associated model
pass db.Model as the parameter for each of the classes
assign tablename with dunders to the name of the table eg: "USERS" and remember caps for the table name
for each attribute, create objects in the class and follow the code from ed to assign appropriate data types, column, primary key and foreign key etc
add necessary comments in the code
Add an item
Activity
Hide Details
Write a comment...
Jordan Aston marked add necessary comments in the code incomplete on this card
just now
Add to card
Members
Labels
Checklist
Dates
Attachment
Custom Fields
Power-Ups
Add Power-Ups
Automation
Add button
Actions
Move
Copy
Make template
Archive
Share


To Do
Doing
Create and normalise ERD. Follow normal form concepts (1NF, 2NF, 3NF) to mitigate duplication and data redundancies.
Planning
Build some mock tables in Google Sheets and pre-fill the fields with sample data.
Planning Coding Ruberic README
QUESTIONS / REMINDERS
Coding
CONTROLLERS
Planning Coding README
AUTHENTICATION / AUTHORISATION
README
README: Create an outline in the readme that includes instructions and requirements for how to run the app.
Coding
MODELS
Coding
SCHEMAS
Cover
SCHEMAS
in list Doing
Labels
Coding
Notifications
Watching
Due date
today at 15:28 due soon
Card Priority
Not Urgent but Important
Description
Edit
Checklist of the steps needed in order to build out the project schems effectively.
Checklist
Hide checked items
Delete
100%
import ma from main.py file
create the Schemas with Marshmallow, it will provide the serialization needed for converting the data into JSON
create the class in each schema, called EntitySchema and pass the parameter ma.Schema
expose the fields - fields object with the attributes assigned in a tuple
implement single schema, when one table needs to be retrieved
implement multiple schema, when many tables need to be retrieved
Add an item
Activity
Hide Details
Write a comment...
Add to card
Members
Labels
Checklist
Dates
Attachment
Custom Fields
Power-Ups
Add Power-Ups
Automation
Add button
Actions
Move
Copy
Make template
Archive
Share


T2A2 - Vinyl Data API
To Do
Ruberic README
R7. Detail any third party services that your app will use.
15 Mar 1 0/5
Ruberic README
R1 / R2. Identify the problem you're trying to solve by building this particular App. Why is it a problem that needs solving?
15 Mar 1 0/7
Coding
CONTROLLERS
13 Mar 15/18
Ruberic README
R8. Describe your project's models in terms of the relationships they have with each other.
15 Mar 1 0/6
Ruberic README
R9. Discuss the database relations to be implemented in the application.
9 Mar 1 0/3
Add a card
Description
Edit
Checklist of the steps needed in order to build out the project controllers effectively.
Checklist
Hide checked items
Delete
83%
import all necessary packages (follow ed)
assign the the entity object to the Blueprint as the top of the code (checked for syntax)
create the decorator with the "GET" method in each controller file.
define a function in each controller file for the GET request, call each function def get_<entity>
build out code for the query inside the function (check ed for syntax)
return the "jsonified" result for GET
create the decorator with the "POST" method in each controller file.
define a function in each controller file for the POST request, call each function def create_<entity>
build out code for the query inside the function (check ed for syntax)
return the "jsonified" result for POST
create the decorator with the "DELETE" method in each controller file.
define a function in each controller file for the DELETE request, call each function def delete_<entity>
build out code for the query inside the function (check ed for syntax)
In the DELETE function there will be lots of auth code commented out, come back and uncomment when ready to implement auth.
remember to handle errors with "abort" descriptions
return the "jsonified" result for DELETE
remember to add comments to all of the controller code.
come back once all of this is implemented and make changes to routes as needed (consider functionality of the app).
Add an item
Dates
Attachment
Custom Fields
Power-Ups
Add Power-Ups
Automation
Add button
Actions
Move
Copy
Make template
Archive
Share

References:

shoyei (2019). Vinyl DJ’s - how do you organize your collection? [online] Available at: https://www.reddit.com/r/DJs/comments/bcimyk/vinyl_djs_how_do_you_organize_your_collection/ [Accessed 12 Mar. 2023].

ceeroSVK (2022). Vinyl djs: Do you catalogize BPM for your tunes somehow? [online] Available at: https://www.reddit.com/r/Beatmatch/comments/vwmhgz/vinyl_djs_do_you_catalogize_bpm_for_your_tunes/ [Accessed 12 Mar. 2023].

MariaDB (2018). ACID Compliance: What It Means and Why You Should Care. [online] MariaDB. Available at: https://mariadb.com/resources/blog/acid-compliance-what-it-means-and-why-you-should-care/.

Contributor, S. (2022). Why Do We Need Object-Relational Mapping? [online] Software Reviews, Opinions, and Tips - DNSstuff. Available at: https://www.dnsstuff.com/why-do-we-need-object-relational-mapping [Accessed 12 Mar. 2023].

Liang, M. (2021). Understanding Object-Relational Mapping: Pros, Cons, and Types. [online] AltexSoft. Available at: https://www.altexsoft.com/blog/object-relational-mapping/.

enqueuezero.com. (n.d.). The Architecture of SQLAlchemy | Enqueue Zero. [online] Available at: https://enqueuezero.com/architecture/sqlalchemy.html#overview [Accessed 16 Mar. 2023].

flask-marshmallow.readthedocs.io. (n.d.). Flask-Marshmallow: Flask + marshmallow for beautiful APIs — Flask-Marshmallow 0.14.0 documentation. [online] Available at: https://flask-marshmallow.readthedocs.io/en/latest/.

www.doppler.com. (n.d.). Using Environment Variables in Python for App Configuration and Secrets. [online] Available at: https://www.doppler.com/blog/environment-variables-in-python [Accessed 16 Mar. 2023].

Patel, H. (2022). Password hashing in Node.js with bcrypt. [online] LogRocket Blog. Available at: https://blog.logrocket.com/password-hashing-node-js-bcrypt/#:~:text=Bcrypt%20is%20a%20library%20to [Accessed 16 Mar. 2023].

flask-jwt-extended.readthedocs.io. (n.d.). Basic Usage — flask-jwt-extended 4.4.4 documentation. [online] Available at: https://flask-jwt-extended.readthedocs.io/en/stable/basic_usage/.