# Information Management & Systems Engineering

## Milestone 2
## Team 27


Adrian Boghean
a11742914

Calin-Nicolae-Virgil Ploscaru
a12142816

# Introduction

A short reminder of what our app is supposed to do. This is taken from Milestone 1.

Our project will store all the relevant information required for a music streaming service

To be able to use a music streaming service, firstly a user has to register. To do so, a user has to sign up using an **e-mail address**. While creating a new account, the users need to also fill in their name and password. Another relevant attribute related to the user is the registration date. This streaming service will also support some sort of social features. For example, a user can follow another user (similar to Spotify's feature). A major difference between the existing streaming platforms and our service is that the users can review/rate albums.

Since it is a music streaming service, our application requires songs. A song is identified by a unique **song id.** Each song has a title, a length, and a release date. While streaming music, the user can like (add to favorites) different songs.

Each song belongs to an album. An album is identified using a unique **album id**. To simplify the problem, we will consider every single release to be an album. This is not far from the reality since most of the single "albums" have another song on the "B-Side". For example, one of the most famous single songs "Wind of Change" by the Scorpions has on the B-Side another song called "Restless Nights". As expected, each album has a name and a release date.

An album is released by an artist. To identify an artist, an **artist id** is used by the platform. Other attributes related to the artist are the artist's name and the artist's musical genre.

To make it easy for the users to discover new music, the platform will show some of the best-rated albums that are available. The album's rating is based on user reviews. Since on this service a review can't be anonymous, a review is identified by the **album id** and by the **user's email**. Each review has to specify a rating between 1 and 5 (5-star rating system) and a text which contains different users' remarks about the album. Another relevant attribute related to the review is the date when it was published.

If a user decides to delete their account, all the reviews are deleted as well. The same happens if an album is removed from the streaming platform.

- ## 2.1.1 Configuration of Infrastructure

To start the container, go to the unzipped folder and run: *$ docker-compose up*

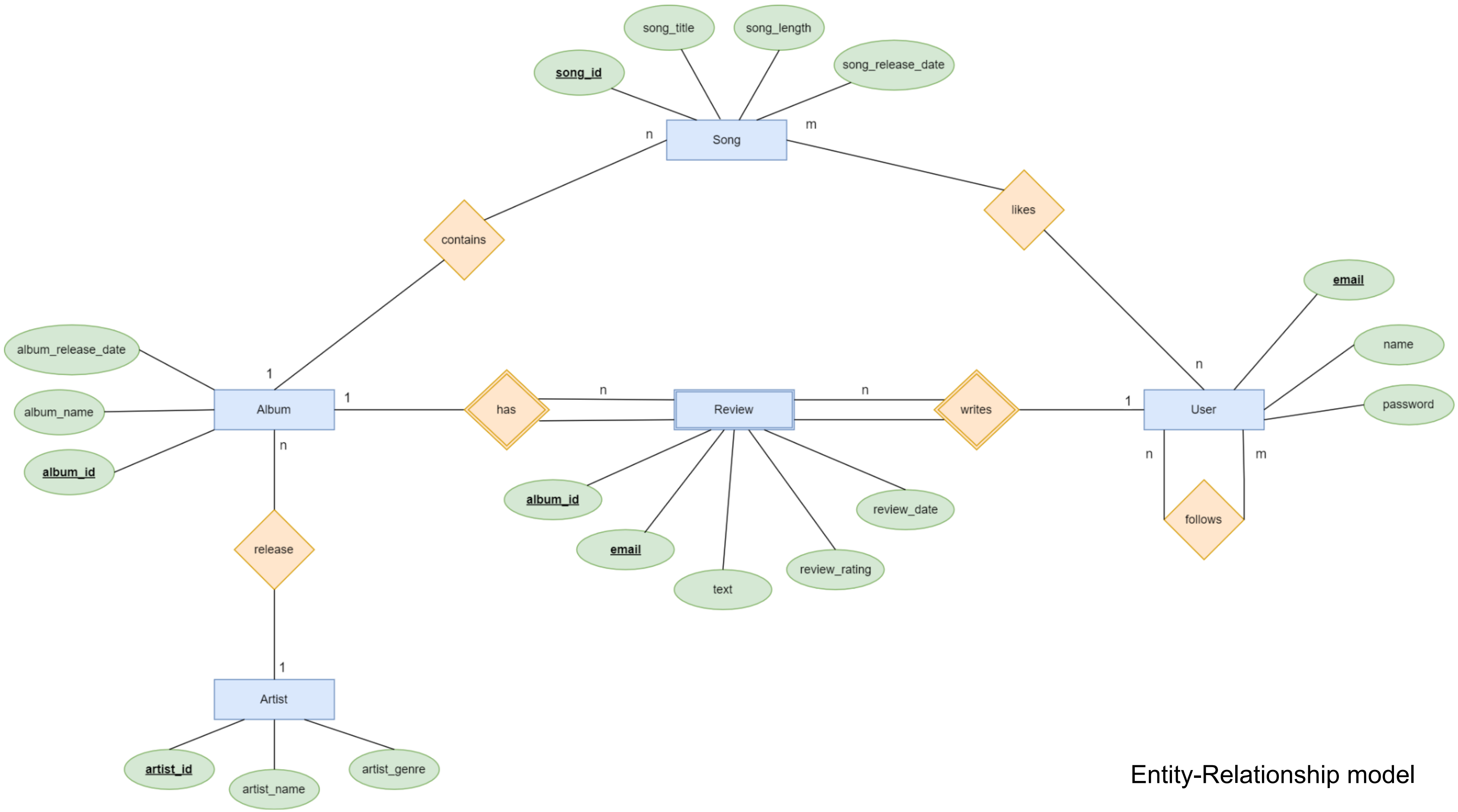
- If the error flask exited with code 1 happens, just wait a bit until mysql and mongodb are finished and then flask should work normally. Flask will automatically retry until it works.
- Then go to [https://127.0.0.1](https://127.0.0.1) and everything should work fine.

Before doing anything, the database has to be initialized.

Containers:

- web - The name of the container is flask, it contains all the backend.
- db - The name of the container is sql, it contains mysql:5.7 image, and it always restarts. The used port for MySQL is 3306.
- **mongodb** - The name of the container is mongo, and it contains the MongoDB database, always restarts and the used port is 27017.
- nginx - container name nginx, always restarts. Uses a self signed certificate.

- **2.1.2 Logical/Physical database design**




Entity-Relationship model

### 2.1.2.1 Logical Database Design of the RDBMS

- Artist (artist_id, artist_name, artist_genre)
    - Candidate Keys: (artist_id)

- Album (album_id, album_name, album_release_date, *artist_id*)
    - Candidate Keys: (album_id)
    - Foreign Key: *album.artist_id* ◇ *artist.artist_id*

- Users (email, username, password, user_registration_date, *followed_by*)
    - Candidate Keys: (email)
    - Foreign Key: *users.followed_by* ◇ *users.email*

- Follows(*email_current, followed_by*)
    - Candidate Keys: (email_current, followed_by)
    - Foreign Key: follows.email_current ◇ *users.email*
    - Foreign Key: follows.followed_by ◇ *users.email*

- Song (song_id, song_title, song_length, song_release_date, *album_id*)
    - Candidate Keys: (song_id)
    - Foreign Key: *song.album_id* ◇ *album.album_id*

- Review (*album_id*, *email*, text, review_rating, review_date)
    - Candidate Keys: (album_id, email)
    - Foreign Key: *review.album_id* ◇ *album.album_id*
    - Foreign Key: *review.email* ◇ *users.email*

- Likes (*song_id*, *email*)
    - Candidate Keys: (song_id, email)
    - Foreign Key: *likes.song_id* ◇ *song.song_id*
    - Foreign Key: *likes.email* ◇ *users.email*

### 2.1.2.2 Physical Database Design of the RDBM

```
CREATE TABLE Artist (
    artist_id INTEGER NOT NULL AUTO_INCREMENT,
    artist_name VARCHAR(50) NOT NULL,
    artist_genre VARCHAR(50),
    PRIMARY KEY (artist_id)
);

CREATE TABLE Album (
    album_id INTEGER NOT NULL AUTO_INCREMENT,
    album_name VARCHAR(50) NOT NULL,
    album_release_date DATE,
    artist_id INTEGER NOT NULL,
    PRIMARY KEY (album_id),
    FOREIGN KEY (artist_id) REFERENCES Artist(artist_id) ON DELETE CASCADE
);

CREATE TABLE Users(
    email VARCHAR(50) NOT NULL,
    username VARCHAR(50) NOT NULL,
    password VARCHAR(50) NOT NULL,
    user_registration_date DATE,
    followed_by VARCHAR(50),
    PRIMARY KEY (email),
    FOREIGN KEY (followed_by) REFERENCES Users(email)
);

CREATE TABLE Follows(
    email_current VARCHAR(50) NOT NULL,
    followed_by VARCHAR(50) NOT NULL,
    FOREIGN KEY (email_current) REFERENCES Users(email) ON DELETE
CASCADE,
    FOREIGN KEY (followed_by) REFERENCES Users(email) ON DELETE
CASCADE,
    PRIMARY KEY (email_current,followed_by)
);
```

```
CREATE TABLE Song(
   song_id INTEGER NOT NULL AUTO_INCREMENT,
   song_title VARCHAR(50) NOT NULL,
   song_length INTEGER,
   song_release_date DATE,
   album_id INTEGER,
   PRIMARY KEY (song_id),
   FOREIGN KEY (album_id) REFERENCES Album(album_id) ON DELETE
CASCADE
);


CREATE TABLE Review(
    album_id INTEGER,
    email VARCHAR(50),
    text VARCHAR(200),
    review_rating INTEGER NOT NULL,
    review_date DATE NOT NULL,
    CHECK (review_rating<=5),
    CHECK (review_rating>=0),
    FOREIGN KEY (album_id) REFERENCES Album(album_id) ON DELETE
CASCADE,
    FOREIGN KEY (email) REFERENCES Users(email),
    PRIMARY KEY (album_id,email)
);

CREATE TABLE Likes(
   song_id INTEGER,
   email VARCHAR(50),
   FOREIGN KEY (email) REFERENCES Users(email) ON DELETE CASCADE,
   FOREIGN KEY (song_id) REFERENCES Song(song_id) ON DELETE CASCADE,
   PRIMARY KEY (song_id,email)
);
```

### 2.1.3 Data Import

To be able to import data, we have decided to use Pandas. Pandas is a Python library which makes data manipulation and analysis easier. Our data for each table is stored as csv files inside */helpers/db_data* folder. Using those csv files and the Pandas library, we have created a function *fill_sql_tables* inside *sql_functions.py* which reads each csv file, parses the rows and the columns. Then it inserts the data into the tables. To observe how it works, flask will display in the terminal the progress of the data importer.

After parsing a document file, the cursor will execute a SQL statement that will look for ‘nan’ data inside the tables, and it will replace the ‘nan’s with **NULL**.

### 2.1.4 Implementation of a Web system

After the website is up and running, the user has to go to : https://127.0.0.1 and press the Initialize DB button. The user is not able to do anything until the database is initialized. After the database is initialized, a message will be displayed to inform the user that the database is initialized. The user can scroll using the navbar through Artists, Songs, Albums, Reviews, Top Albums by each Artist and The Most Reviewed Albums. Additionally, the user can migrate the SQL database to NoSQL and also completely reset the website. Other than that, the user can also log in or logout. On the Albums page, the user may post a review. To do so, the user has to be logged in. A user to test how it works is:

Email: adrian@gmail.com
Password: 123aaa123

Besides that, a new account can be created from the login page.

After the user is logged in, posting a review is possible. To post a review, an album name is required. If the album is not found, the user will be informed that the albums does not exist in our database. If the user has already reviewed that album, the review will be updated and the user will be informed that the review was updated. In the MongoDB tab, the user can decide to Migrate to NoSQL. This should take no longer than 10 seconds. Anyway, the user will be informed when the migration is done. After the migration happen, everything should work like on the SQL version. If the user decides to do the initialization again, he may press the ResetWebsite button and everything will start once again with an empty database.

To connect to the mongodb database:
mongodb://user:password@localhost:27017/?authMechanism=DEFAULT

## 2.1.5 Implementation of Use Cases and Reports

**Report: Each artist's top-rated album. (Adrian Boghean)**

- **Description:** This report will display for each artist the best-rated album. When a user posts a review, the user has to give a rating to the album. The top-rated album for each artist is the one with the highest average user score. To do so, the report will filter all albums by artist and find the one with the best rating.
- **Entities:** Album, Artist, Review
- **Filter by**: Artist
- **Sort by:** Review Rating

### *SQL*

```
SELECT *
FROM (SELECT Review.album_id, Artist.artist_id, Artist.artist_name, Album.album_name,
avg(review_rating) As averageRating
FROM Review
LEFT JOIN Album ON Album.album_id = Review.album_id
LEFT JOIN Artist ON Artist.artist_id = Album.artist_id
GROUP BY Artist.artist_id, Review.album_id) a WHERE NOT EXISTS (
        SELECT * FROM (SELECT Review.album_id, Artist.artist_id, Artist.artist_name,
avg(review_rating) As averageRating
        FROM Review
        LEFT JOIN Album ON Album.album_id = Review.album_id
        LEFT JOIN Artist ON Artist.artist_id = Album.artist_id
        GROUP BY Artist.artist_id, Review.album_id) b WHERE a.artist_id = b.artist_id AND
a.averageRating < b.averageRating )
   ORDER BY averageRating DESC
```

*NoSQL*

```
pipeline = [
    {
        '$lookup': {
            'from': 'review',
            'localField': 'album_id',
            'foreignField': 'album_id',
            'as': 'reviews_new'
        }
    },

    {'$unwind': "$reviews_new"},
    {
        '$lookup': {
            'from': 'artists',
            'localField': 'artist_id',
            'foreignField': 'artist_id',
            'as': 'artist_id2'
        }
    },
    {'$unwind': "$artist_id2"},

    {
        '$project': {
            "_id": 0,
            "album_name": 1,
            "artist_name": '$artist_id2.artist_name',
            "review_rating": '$reviews_new.review_rating'

        }
    }
]
df = pd.DataFrame(list(mongo_db['albums'].aggregate(pipeline)))

cols = ['album_name', 'artist_name']
df2 = df.groupby(cols)['review_rating'].mean().reset_index()
df2 = df2.sort_values(by='review_rating',ascending=False)
df2 = df2.drop_duplicates(subset=['artist_name'], keep = "first")
```

After creating the DataFrame using the pipeline from above, I was able to directly manipulate it using Python. Associating duplicated data with values from another column and then keeping only the data with the highest value from the other column was done much easier using pymongo and pandas than using SQL.

On the SQL statement, I have started from the Review table and LEFT Joined the Album table only when the album_id from the Album table matches the one from the Review table. The reason is, that if I need to find each artist's top-rated album, I need to look only at the albums that have reviews. Then, to find out who is the artist of each's album, I have LEFT Joined the Artist table as well. The LEFT Join happens only when the album's artist_id matches the artist_id from the Artist table. Then, I have grouped the "selected" new table by artist_id and review.album_id. The resulting table I have named it ***a*** and then I have done the exactly same selected and named the resulting table ***b***. I have done that to be able to compare the average rating and to remove the albums that don't have the highest average rating. Of course, that will happen only when a.artist_id = b.artist_id. To put the result of the a and b tables, I have used a SELECT …. WHERE NOT EXISTS (....) a.artist_id = b.artist_id AND a.averageRating < b.averageRating.

On the NoSQL statement, I have used a pipeline statement. The $lookup works like a join. I have started from the album collection this time and I have looked up in the Review collection for reviews that have the same album_id as the ones from the album collection. Then, I have looked up in the artist collection to get the information of the artists that have albums reviewed. Then, I have used $project to decide which Field I need in my final Collection. After that I have used the pymongo and pandas to manipulate the dataframe. Firstly, I have grouped the ['album_name', 'artist_name'] columns and calculated the average rating of each album. Then, since I was left with all the albums that were reviewed from an artist, I had to sort them by the highest average review rating, and then I have removed the duplicated from the ['album_name'] just by keeping the first one. That was possible because I have sorted the column before. After grouping the columns of the data frames, I also had to reset the indexes so that I could be able to iterate once again in the data frames.

**Report: Albums with the most reviews from accounts created last year (Calin Ploscaru)**

- **Description**: This report aims to find out the albums that have had the most reviews from new accounts that have been created last year. In order to achieve this, we get the albums that have had the most reviews and then we filter them out by the account's registration date.
- **Entities**: Album, User, Review
- **Filter by**: User
- **Sort by**: Review count

## *SQL*

```
SELECT Album.album_name, COUNT(Review.album_id) as review_count
FROM Album
LEFT JOIN Review ON Album.album_id = Review.album_id
LEFT JOIN Users ON Review.email = Users.email
WHERE YEAR(Users.user_registration_date) = YEAR(NOW()) - 1
GROUP BY Album.album_name
ORDER BY review_count DESC
```

## *NoSQL*

```
pipeline = [
        {
            '$lookup': {
                'from': 'review',
                'localField': 'album_id',
                'foreignField': 'album_id',
                'as': 'reviews_new'
            }
        }, {
            '$unwind': '$reviews_new'
        }, {
            '$lookup': {
                'from': 'users',
                'localField': 'reviews_new.email',
                'foreignField': 'email',
                'as': 'users_new'
            }
        }, {
            '$unwind': '$users_new'
        }, {
            '$match': {
                'users_new.user_registration_date': {
                    '$gte': datetime.datetime(2021, 1, 1, 0, 0, 0, tzinfo=datetime.timezone.utc),
```

```
                '$lte': datetime.datetime(2021, 12, 31, 23, 59, 59,
tzinfo=datetime.timezone.utc)
              }
            }
          }, {
            '$group': {
              '_id': {
                'album_name': '$album_name'
              },
              'review_count': {
                '$sum': 1
              }
            }
          }, {
            '$sort': {
              'review_count': -1
            }
          }
        ]
        df = pd.DataFrame(list(mongo_db['albums'].aggregate(pipeline)))
        results = df.values
        return render_template("mostreviews.html", data=results, mongoyes="MongoDB
results")
```

On this SQL Statement, I have first selected the attribute album_name from the table Album and then counted and selected the number of reviews each album had from the table Review with the COUNT function and gave it the ‘review_count’ alias. Then, I have first joined the Album table with the Review table on their respective album_id keys with a LEFT JOIN function in order to join the Users table on their email keys with another LEFT JOIN function. This is because we will need the Users table to filter out the users who have joined last year. In order to filter them out, I added a WHERE clause where we check if the user’s registration date has been created last year or not. We then group the results by the Album name and in order to show which albums have the most reviews, we just order it by review_count in descending order.

On the NoSQL statement, I have first joined the ‘albums’ collection with the review collection with the $lookup operator and joined them on the ‘album_id’ that each of them had, and we have given this joined collection the name ‘reviews_new’. Afterwards, I have joined the ‘reviews_new’ collection on a new collection called ‘users_new’ on the field ‘email’. This is so that we can manage to filter out the reviews by the users. We then used a $match operator to filter out the user’s registration date from the documents. We then used a $group operator to group the results by the album name and while we are grouping it, we count the reviews that each album has. We then do a $sort operation that sorts the reviews in descending order. We then use pandas and dataframes to print out the results into the template we’ve given it.

## 2.2 NoSQL design decisions

For the NoSQL part, we have tried to implement the design by using ‘pandas’ and DataFrames. We wanted to use pandas in order to add subcollections to our collections in order to satisfy the relationships in the schema. Unfortunately, we weren’t able to fully implement the NoSQL design for the RDBMS. However, we did have in mind a plan on how we could have implemented the schema from before in MongoDB. We thought of creating 4 collections at first, the first ones being the Album, Artist, Users and Song tables as the main collections and the rest of the tables as subcollections for those tables.

### 2.2.1. Album

The Album collection would inherit the exact same attributes as in the SQL Database but with the NoSQL design, we intended to implement the relationships they had with the other tables in the RDBMS as subcollections. But the Album table in the schema has 3 one-to-many relationships with the other tables in the schema and implementing them all would take us a lot of disk space and updating the collections would be very expensive. So for the Album table, for the purpose of this project, would have only one relationship implemented which is the one it has with Review. The Album collection would have a subcollection of reviews, each review having the exact same attributes except the album_id attribute. This is because it would have been very useful for the first use case in which the user would need to post a review for an album that he wants. Each album will have multiple reviews from different users, but not the same review from the same user.

For indexing on this collection, we have used album_id but also artist_id. They are both required to do the first report (Each artist’s top-rated album). Album_id is also required to do the first main case, posting a review to an album.

### 2.2.2. Artist

The Artist collection would be the exact same as in the previous database. There is no implementation of any relationships because they aren’t beneficial for the purpose of this project as it would just waste more disk space.

For indexing, we have used the artist_id since it is required for the use-cases. Each artist’s top-rated album report needs it too.

### 2.2.3. User

The User collection is similar to the SQL variant, it has the same attributes but the collection would have had more enhancements done to it. The User collection would have had all of its relationships implemented, the first of which is the relationship with the Review table. Instead of having review as a table, reviews will be added as a subcollection to User, same as with the Album collection. This is so that we can make NoSQL queries much easier to make with each subcollection having the same attributes but without the email attribute. Then there's the relationship with the Follows table, which is a many-to-many (M:N) relationship. The way it would have been implemented would be like this: Each user collection would have a subcollection of users called follows which is supposed to represent the users that each user is following. Each user would have been followed by a lot of other users and some users themselves would have been followed by the users they are following. The last relationship that would have been implemented would be the M:N relationship which the Songs table. A Likes subcollection would have been added to User which represents the songs that each users likes.

For indexing, we have used the email. Email is required for sessions, for adding a review but also for displaying the liked songs and of course for the login system.

### 2.2.4. Song

Last but not least, the Song table has the same attributes but with one single enhancement, that enhancement being its relationship with the User table. As mentioned before, every user has a likes subcollection which represents the songs that that specific user likes. The song collection will have one more subcollection that will be referenced from User which is the 'liked' subcollection. This subcollection represents the users which have liked that specific song.

For indexing, we have used the song_id but also the album_id. When a user likes a songs(first use case), firstly the use has to search for a song (second use case). When the user finds a song and wants to like it, the song_id has to be stored so that it can be inserted in the 'likes' collection.

| Starting Date | End Date | Hours | Who | Activity | Result |
|---|---|---|---|---|---|
| 3.05.2022 | 3.05.2022 | 2.5 | Adrian + Calin | Watched the Docker tutorial, created basic containers for what is needed. | Docker works. |
| 4.05.2022 | 4.05.2022 | 2 | Adrian | Created a basic Flask application to see how it works | A hello world flask |
| 6.05.2022 | 6.05.2022 | 1 | Adrian | Connected MySQL and MongoDB to Python | Connections between containers. |
| 6.05.2022 | 6.05.2022 | 1 | Adrian | SQL Statement to create tables in Python | Tables in the MySQL database. |
| 8.05.2022 | 8.05.2022 | 3.5 | Adrian | Created sample data and the filler. Added some basic functionality to the Python app: create, delete and reset database. | We have data in our MySQL database. |
| 12.05.2022 | 12.05.2022 | 0.5 | Adrian | Fixed some problems with our initial tables. | Tables in MySQL database fixed. |
| 17.05.2022 | 17.05.2022 | 2 | Adrian | Started writing the Report | Some information in the Report + Logical/Physical database design |
| | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| 17.05.2022 | 17.05.2022 | 1 | Adrian | Added bootstrap, created a navigation bar and changed some design elements of the website using CSS. | A better looking website. |
| 21.05.2022 | 21.05.2022 | 1.5 | Adrian | Fixed some bugs and added button to initialize the database. | Website works better, to do anything the database has to be initialized firstly. |
| 30.05.2022 | 30.05.2022 | 1.5 | Adrian | Done Report 1, Each artist's top-rated album | Top-rated albums by each artist added on the website. |
| 4.06.2022 | 4.06.2022 | 0.5 | Adrian | Improved the DB Filler, now it adds more data and implemented some sort of workaround to avoid duplicated primary keys when generating random data. | More data. |
| 4.06.2022 | 4.06.2022 | 1.5 | Adrian | Implemented my main use case: Posting a review | Logged users can now post reviews to the existing albums. |
| 12.06.2022 | 12.06.2022 | 2 | Adrian | Watched the tutorial about MongoDB and looked online what do we need in Python so that we can communicate with our | I've got an idea how it works. |

| | | | | | |
|---|---|---|---|---|---|
| | | | | MongoDB | |
| 14.06.2022 | 14.06.2022 | 4.5 | Adrian | Added required library. Tested pymongo for the first time to see how it works. Learned about new methods from pandas library. | Artist, Album, Users and Song tables are now collections in MongoDB. |
| 15.06.2022 | 15.06.2022 | 1.5 | Adrian | Migration of all SQL tables completed. Now the website lists the MongoDB collections. | MongoDB collections are visible on the website now. There is a new button to migrate and also a new button to reset the website. Use cases no longer work and also the reports at this point. |
| 16.06.2022 | 16.06.2022 | 2 | Adrian | Adapted the add review method for MongoDB. | Users can now post reviews using the MongoDB database. |
| 18.06.2022 | 18.06.2022 | 2.5 | Adrian | Report 1 in MongoDB | Top-rated albums by each artist using the MongoDB database |
| 18.06.2022 | 18.06.2022 | 1 | Adrian | Wrote information in this report. | Added and described the SQL and the NoSQL statements in the report. |
| 19.06.2022 | 19.06.2022 | 0.5 | Adrian | Improved the CSS a bit, so the tables looks better, and the data is well aligned. | Web looks better. |
| 19.06.2022 | 19.06.2022 | 1.5 | Adrian | Added a my songs section for NoSQL and also SQL | The logged-in user can see the liked/favorite songs |

| | | | | | |
|---|---|---|---|---|---|
| 19.06.2022 | 19.06.2022 | 0.5 | Adrian | Wrote in this report | Added more information to 2.1.4 |
| 21.06.2022 | 21.06.2022 | 2 | Adrian | After searching for a song, a user can now add the founded song to the favourites | A logged-in user can add new items in theirs favourite list. |
| 21.06.2022 | 21.06.2022 | 1 | Adrian | Added self-signed certificate to docker nginx:alpine | https://127.0.0.1/ can be accessed but "CA Root certificate is not trusted because it is not in the Trusted Root Certification Authorities store" because the certificate is self-signed |

| | | | | | |
|---|---|---|---|---|---|
| 18.05.2022 | 18.05.2022 | 1.5 | Calin | Added pages to the navigation bar, created the routes to the templates in the application and added queries to test the tables | Routing, the HTML pages and SQL queries work |
| 31.05.2022 | 31.05.2022 | 1 | Calin | Done report 2. Albums with the most reviews from accounts created last year | Report successfully shown on the website |
| 03.06.2022 | 03.06.2022 | 2.5 | Calin | Added login, logout pages. Created a register page as a | Data gets successfully inserted into the SQL |

| | | | | | |
|---|---|---|---|---|---|
| | | | | main use case. | Database and login works as well. |
| 04.06.2022 | 04.06.2022 | 0.5 | Calin | Added permanent sessions for users to stay logged in | Users can now stay logged in |
| 17.06.2022 | 17.06.2022 | 1.5 | Calin | Made login, logout and register pages work for MongoDB | Login system works in MongoDB |
| 20.06.2022 | 20.06.2022 | 2 | Calin | Made report 2 work for MongoDB | Report 2 migration successfully completed. |

Sources:

For flask, we have used this: Flask Tutorial #1 - How to Make Websites with Python
For pandas: (they are mentioned in the code as well)

- https://pandas.pydata.org/docs/reference/api/pandas.read_sql_table.html
- https://blog.panoply.io/how-to-read-a-sql-query-into-a-pandas-dataframe
- https://stackoverflow.com/questions/49221550/pandas-insert-a-dataframe-to-mongo

For the pipeline:
https://www.stackchief.com/tutorials/%24lookup%20Examples%20%7C%20MongoDB