Documentation

Requirements Specification

The Goal of this application is to create a database for a streaming service (like Netflix). Therefore a database is created, which contains movie informations, informations about the involved directors, actors and the used genres. Also there should be several users, which are in this case all admin users, who can create new movies, actors, etc. and delete them.

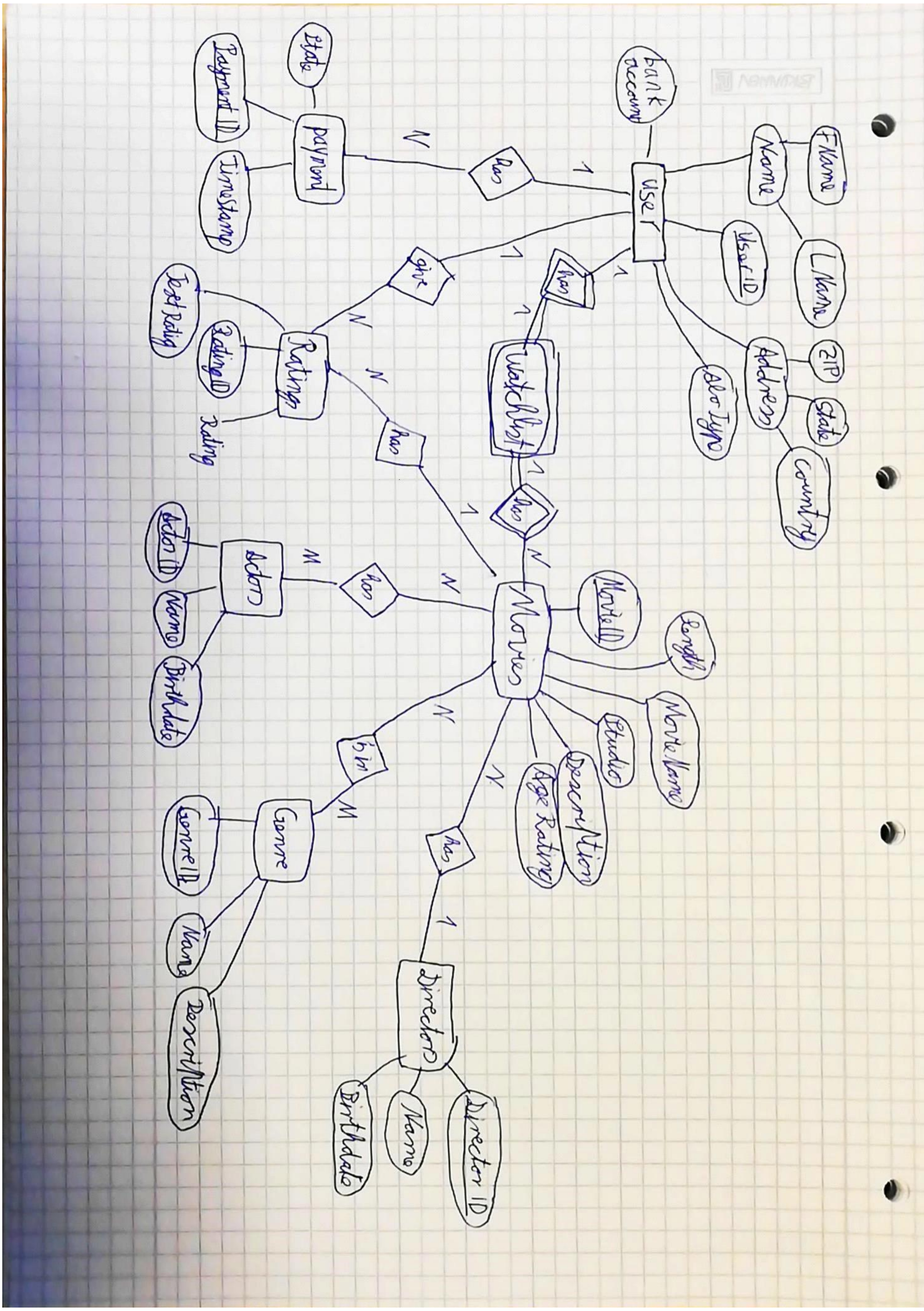
The database will contain these tables:

- User_table
- Payments
    - Not in real use for this project, but can be used as soon as “real” consumers uses the service to check if they paid the bill
- Watchlist
- Movies
- Actors
- Directors
- Genre
- Ratings

For a more specified definition of the tables check the files:

- Table Definition.pdf
- Table Description.docx

ER-Diagram



Notes:

- The relation from watchlist to movies is n:m instead of 1:n

    - Attributes of has: MovieID, WatchlistID
- Between movies and Genre and movies and actors there are attributes missing
    - MovieID and ActorID / GenreID

Exercise 3

3.1 Normalization

1. Normalform: eine Relation ist in der ersten Normalform, wenn jedes Attribut atomare Werte hat und jeder Wert aller Attribute in einem Tupel einzigartig ist.

Diese Bedingung ist erfüllt, da jedes Attribut nicht weiter zerteilbare Werte hat. Zudem wird über die Keys gewährleistet, dass jedes Tupel einzigartig ist.

2. Normalform: eine Relation ist in der zweiten Normalform, wenn sie sich in der ersten Normalform befindet und jedes Nichtschlüsselattribut vom Primärschlüssel abhängt und nicht von einem Teil des zusammengesetzten Schlüssels.

Diese Bedingung ist erfüllt, da alle Nichtschlüsselattribute vom Primärschlüssel abhängen. Zudem ist es auch in der ersten Normalform, die bereits überprüft wurde.

3. Normalform: Eine Relation ist in der dritten Normalform, wenn sie in der zweiten Normalform ist und kein Attribut hat, das nicht vom Primärschlüssel transitiv abhängt.

Diese Bedingung ist erfüllt, da jedes Attribut vom Primärschlüssel transitiv abhängt. Zudem ist es auch in der zweiten Normalform, die bereits überprüft wurde.

See also: here

3.2 Create SQL-Queries

```
/* Create all Tables */
CREATE TABLE IF NOT EXISTS User_Table(
        F_Name                  VARCHAR(30),
        L_Name                  VARCHAR(30),
        UserID                  INT PRIMARY KEY,
        Zip_Code                INT,
        City                    VARCHAR(20),
        Home_State              VARCHAR(30),
        Country                 VARCHAR(20),
        eMail                   VARCHAR(30),
        BIRTHDATE               DATE,
        AboType                 VARCHAR(10),
        BankAccount             VARCHAR(30)
);

CREATE TABLE IF NOT EXISTS Payment(
        Payment_ID              INT PRIMARY KEY,
        UserID                  INT,
        TimeStamp               DATE,
        Actual_State        BOOLEAN,
        FOREIGN KEY (UserID) REFERENCES User_Table(UserID)
);

CREATE TABLE IF NOT EXISTS Genre(
        GenreID                 INT PRIMARY KEY,
        GenreName               VARCHAR(20),
```

```
        GenreDescription VARCHAR(50)
);

CREATE TABLE IF NOT EXISTS Actors(
        ActorID                 INT PRIMARY KEY,
        ActorName               VARCHAR(30),
        Birthdate               DATE,
        MovieIDs        INT
);

CREATE TABLE IF NOT EXISTS Directors(
        DirectorID      INT PRIMARY KEY,
        DirectorName    VARCHAR(30),
        Birthdate       DATE,
        MovieIDs        INT
);

CREATE TABLE IF NOT EXISTS Movies(
        MovieID         INT PRIMARY KEY,
        MovieLength             INT,
        MovieName               VARCHAR(30),
        GenreID                 INT,
        DirectorID              INT,
        ActorID                 INT,
        Studio                  VARCHAR(20),
        Description             VARCHAR(100),
        AgeRating               INT,
        FOREIGN KEY (GenreID) REFERENCES Genre(GenreID),
        FOREIGN KEY (DirectorID) REFERENCES Directors(DirectorID),
        FOREIGN KEY (ActorID) REFERENCES Actors(ActorID)
);

CREATE TABLE IF NOT EXISTS Watchlist(
        UserID                  INT,
        MovieID                 INT,
        Foreign KEY (UserID) REFERENCES User_Table(UserID),
        FOREIGN KEY (MovieID) REFERENCES Movies(MovieID)
);

CREATE TABLE IF NOT EXISTS Ratings(
        RatingID        INT PRIMARY KEY,
        MovieID                 INT,
        UserID                  INT,
        RATING                  INT,
        TextRating      VARCHAR(100),
        FOREIGN KEY (MovieID) REFERENCES Movies(MovieID),
        FOREIGN KEY (UserID) REFERENCES User_Table(UserID)
);

/* Add Foreign Key to Tables who reference each other*/
ALTER TABLE Actors ADD FOREIGN KEY (MovieIDs) REFERENCES Movies(MovieID);
ALTER TABLE Directors ADD FOREIGN KEY (MovieIDs) REFERENCES Movies(MovieID);

/*Add some data and delete old data */

TRUNCATE Actors CASCADE;
TRUNCATE Directors CASCADE;
TRUNCATE Genre CASCADE;
TRUNCATE Movies CASCADE;
TRUNCATE Payment CASCADE;
TRUNCATE User_Table CASCADE;
TRUNCATE Watchlist CASCADE;
```

This SQL Code is not used, due to the reason that we create the database with SQL-Alchemy!
See real SQL code: here

3.3 Index on Database

```
CREATE INDEX actor_names ON actors (ActorName);

CREATE INDEX movie_basic_data
ON movies (MovieName, MovieLength, MovieDescription);

CREATE INDEX user_login_credentials ON user_table (eMail,
birthdate);
```

See also: here

3.4 View on Database

```
CREATE VIEW Movie_with_Ratings AS
SELECT m.MovieName, r.Rating, r.TextRating FROM movies m, ratings r
WHERE m.MovieID = r.MovieID
```

See also: here

3.5 / 3.6 Insert SQL-Queries

Some example inputs.
These inputs are not used, due to implementation via SQL-Alchemy → See later in Documentation.

```
INSERT INTO Actors(
        actorid, actorname, birthdate, movieids)
        VALUES (1, 'Peter Dinklage', '07.11.1975', null),
                        (2, 'Sophie Turner', '02.22.1981', null),
                        (3, 'Emilia Clarke', '07.02.1991', null);

INSERT INTO Directors(
        directorid, directorname, birthdate, movieids)
        VALUES (1, 'Denies Villneuv', '01.01.1999', null),
                        (2, 'David Cameron', '09.19.1956', null),
                        (3, 'Rian Coogler', '12.24.1989', null);

INSERT INTO Genre(
        genreid, genrename, genredescription)
        VALUES (1, 'Action', 'Something explodes.'),
                        (2, 'Drama', 'Really dramatic.'),
                        (3, 'Comedy', 'Wanna laugh?');

INSERT INTO User_Table(
        f_name, l_name, userid, zip_code, city, home_state, country, email, birthdate, abotype, bankaccount)
        VALUES ('Marie', 'Sanders', 1, 1234, 'Boston', 'Texas', 'U.S.A.', 'marie@sanders.com', '03.09.1971', 'Test', '5505 8765 9863'),
                        ('Jack', 'Rhino', 2, 5643, 'Huston', 'Nevada', 'U.S.A.', 'jack@rhino.net', '08.19.1971', 'Normal', '5505 0076
1233'),
                        ('Tony', 'Stark', 3, 7753, 'Paris', 'Ile de Paris', 'France', 'tony.stark@avengers.com', '11.22.1991', 'Premium',
'1123 0854 6466');

INSERT INTO Payment(
        payment_id, userid, "timestamp", act_state)
        VALUES (1, 2, '01.01.2020', true),
                        (2, 2, '02.01.2020', false),
                        (3, 3, '02.01.2020', true);

INSERT INTO Movies(
```

```
	movieid, movielength, moviename, genreid, directorid, actorid, studio, description, agerating)
	VALUES (1, 123, 'Sharknado: The Return', 1, 2, 2, 'Warner Bros', 'Experience the return of the biggest thread to humanity', 12),
			(2, 164, 'Space Quest: The new Hope', 2, 1, 3, 'Disney', 'Only as a Team it will be possible to defeat the
death moon.', 16),
			(3, 94, 'Laughing: Out Loud', 3, 3, 1, 'Sony', 'See the Origin of LoL.', 6);

INSERT INTO Watchlist(
	userid, movieid)
	VALUES (1, 1),
			(2, 1),
			(2, 3);
```

See also: here

3.7 Several SQL-Queries

1. Search the director of every movie ordered by the actor ID

```
select*
from movies join actors

on movies.actorid = actors.actorid order by movies.actorid asc;
```

2. Search every movie that has the genre ID 2 and a length above 100 minutes

```
select*
from movies where genreid = 2

intersect

select *
from movies
where movielength > 100
```

3. Search every user who is in the table payment

```
select*
from user_table
where (userid) in (select userid

from payment );
```

4. Count the number of movies by of every genre

```
select genreid , count(*) movieid from movies*
group by genreid
```

5. Search every user and his rating if he has one

```
select *
from user_table left join ratings

on user_table.userid = ratings.userid where user_table.userid = Null;
```

6. Delete user with ID = 1 from table user_table

```
Delete from user_table Where userid = 1;
```

7. Search the moviename of every movie produced by Disney

```
select moviename
from movies
where studio like 'Disney';
```

See also: here

Start the application

- Make sure you have Docker Desktop installed and running
- Navigate to databases/databaseproject/Dockersetup via Terminal
- Run (sudo) docker-compose build | can take a lot of time (use sudo in case of error)
- Run sudo docker-compose up | can take also several minutes
- Example Login credentials: marie@sanders.com, 1971-03-09

Application Description

Frontend

The Application uses a react frontend.
The users accesses the application via localhost:3000.

From this site the user has the possibility to login to the streaming service with login credentials, or create a new user to use the service.
The login masque uses the user-email and the birthdate as password.

After the login, the user is directed to the main page, from where he can access every part of the application.

He can alter via a dropdown-menu the actors, directors and movies or create new instances of them. Therefore the user is directed to a seperat page, where he gets all information about the movie / actor / genre stored in the db. From there he can edit the informations or delete the item.
To delete something the user has to verify the process with a second button.

From the Mainpage the user can also edit his personal user informations or delete his account. The Deletion will automatically logout the user.

From the Mainpage there are also the Movielist and Watchlist accessible.
The Watchlist contains all the movies the user added to the list.
From both lists the user can access the Movie Informations.
Here he gets all information stored about the movie, can add / delete it from his watchlist or give a rating to this movie.
From the Movielist Site the user is also able to add a new movie to the database.

Restrictions:

- There are no real movie files stored in the application / database, so you can't watch a movie!

- Movie-Information can't be updated, due to the reason that there is no logical reason, that these informations can change. When a movie is finished there won't be new actors in it afterwards and the old actors can't be cut out, for example

Application Information:
- Sometimes the application needs some time to start or load pages
    - Please give it some time or reload the whole page if there is an error. This could happen sometimes due to connection to backend isn't very fast.

Backend

The backend is implanted in flask and accesed via axios from react.
Since there are problems in axios for asynchronous connection we implemented a very dirty connection with axios (stacked axios call to delay the setState function). This is not very clean, but one possible way to implement the flask backend with axios.

The flask backend connects via the config.py file with the postgres database in another docker container. If you want to run the application on your local machine, without docker, you have to update the local DATABASE_URI String in the config.py file and set your local username and password!

The backend waits for an axios call on a given route and then gets the data via sql-alchemy.

See Backend: here (../databasesproject/api/api.py)

Database

The database is implemented with SQLAlchemy in Python and gets the entries via yaml-files.

The Models are created in this file: here (../databasesproject/api/models.py)

The Insert commands are in this file: here (../databasesproject/api/crud.py)
We used yaml files to insert the data, because this is a very clean and readable way to insert base data to the database.
Because it wasn't possible to set up the n:m relations in this way there are several commands at the end of the file, that populates this relations.

Docker

We used Docker to get the application run on all systems.
Therfore we set up a docker container for:
- The database
- The flask-backend
- The react-frontend

The Docker-Compose file orchestrates all containers. (../databasesproject/Dockersetup)