V.01

DESIGN DOCUMENT

NETFLIX MANAGER

BLOSSOM ANUKPOSI, CRISTIAN TRIFAN, STEFAN CAZACU,
MIKHAIL JOSAN

GROUP 11 | Netflix Management API

INTRODUCTION

This document aims to cover aspects of the Netflix Management API that have not already been covered by the extensive Swagger Documentation. This will include information about the design of the different modules of the system, and how they interact with each other.

ARCHITECTURE SLAB

On a very basic level, the architectural overview includes the internal API which contains several endpoints for a list of entities including Account, Episode, Genre, Movie, Preference, Profile, Referral Discount, Season, Series, Subscription, Subtitle, Watched Media List, and Watchlist.

The full documentation of what endpoints influence these entities can be found on localhost:3000/api-docs/#

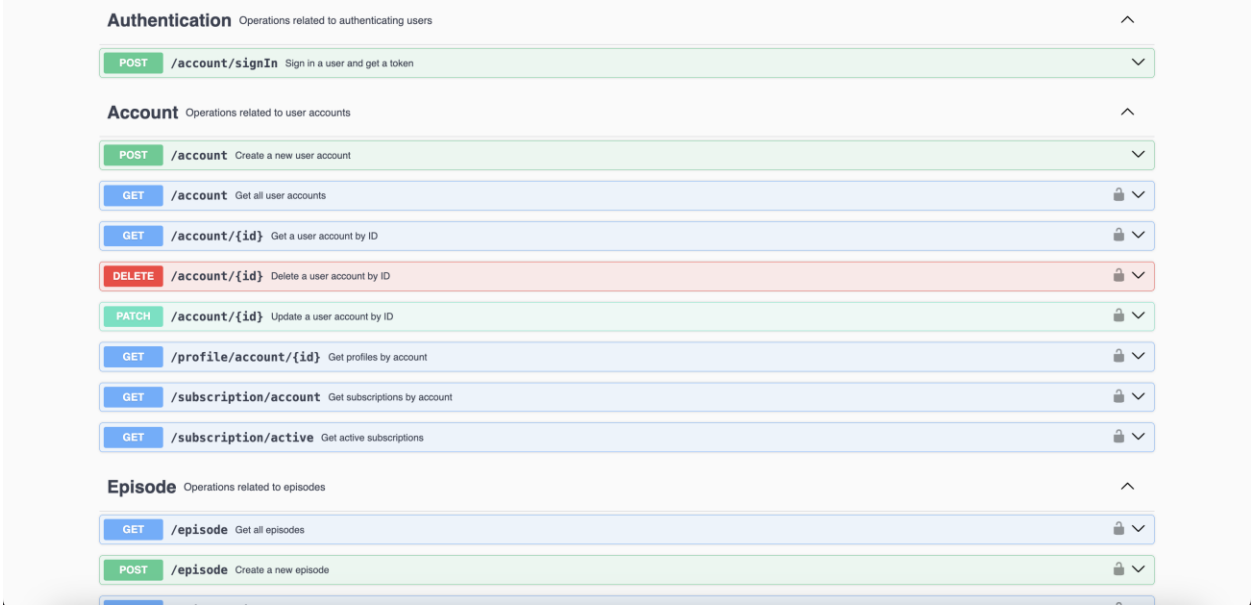
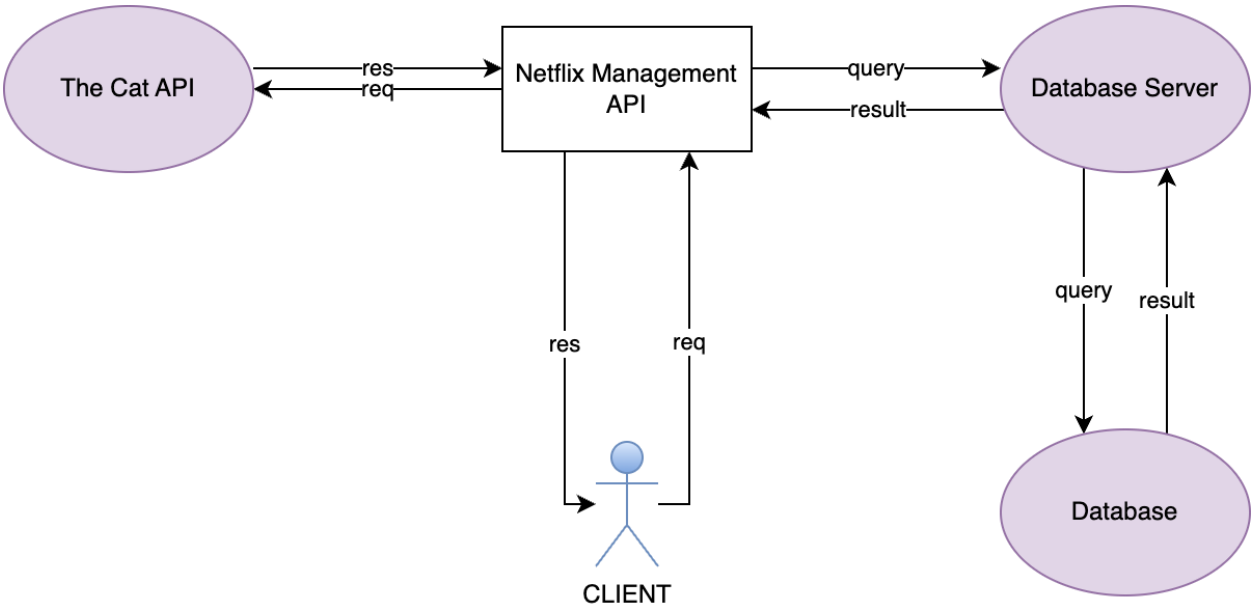


Image of Swagger documentation

For all endpoints, we have basic GET (allEntries), GET (entryById), POST (newEntry), and DELETE (entryById). Some accounts have special endpoints like GET profilesByAccount or GET episodesBySeason. An example of how they work is well documented on Swagger docs.

At its base, the Netflix Management API engages several parts.

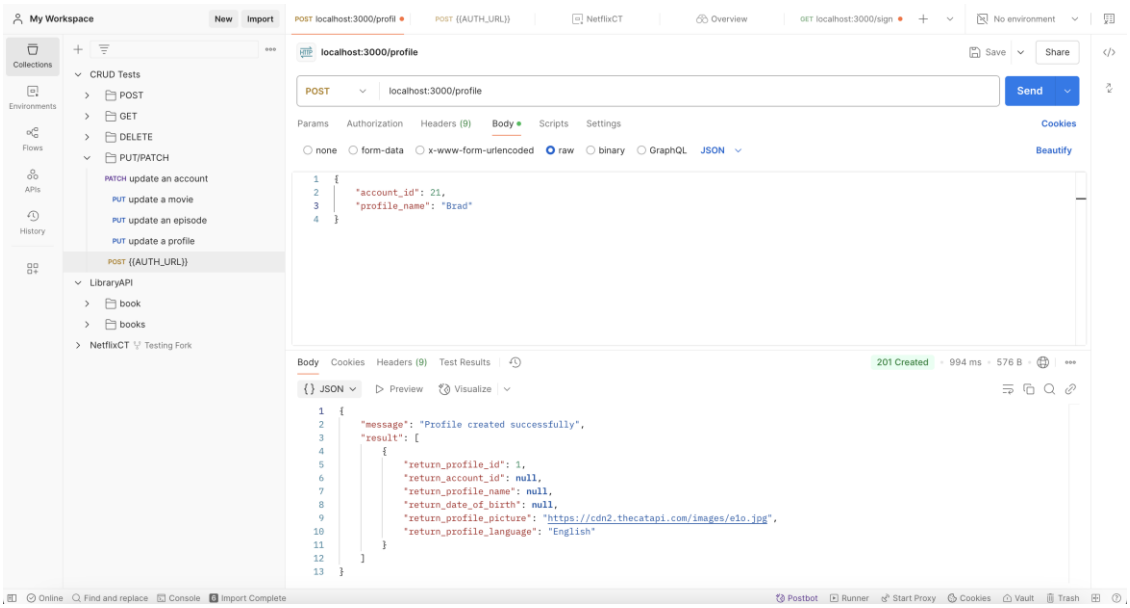


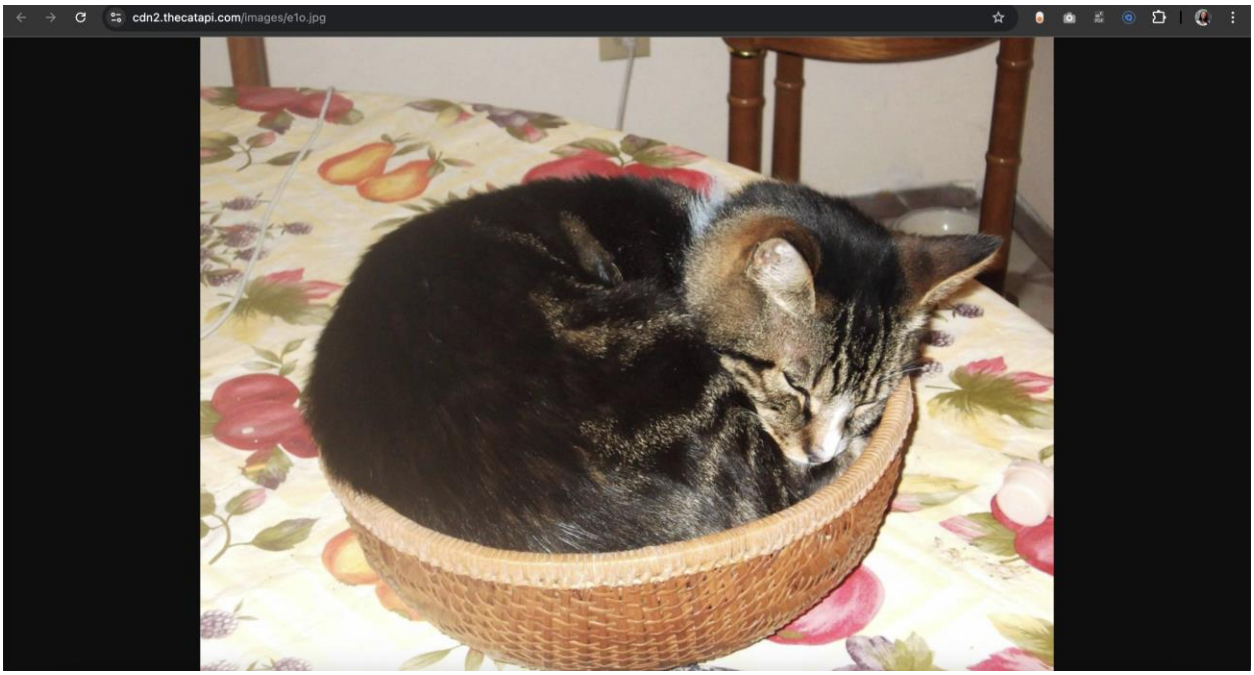
The structure of the API

The Netflix API communicates with an external API called The Cat API. It generates random cat images and assigns them to profiles that are created with a [null] profile picture value. https://thecatapi.com/

An example of how this looks like is this:

Request and response:



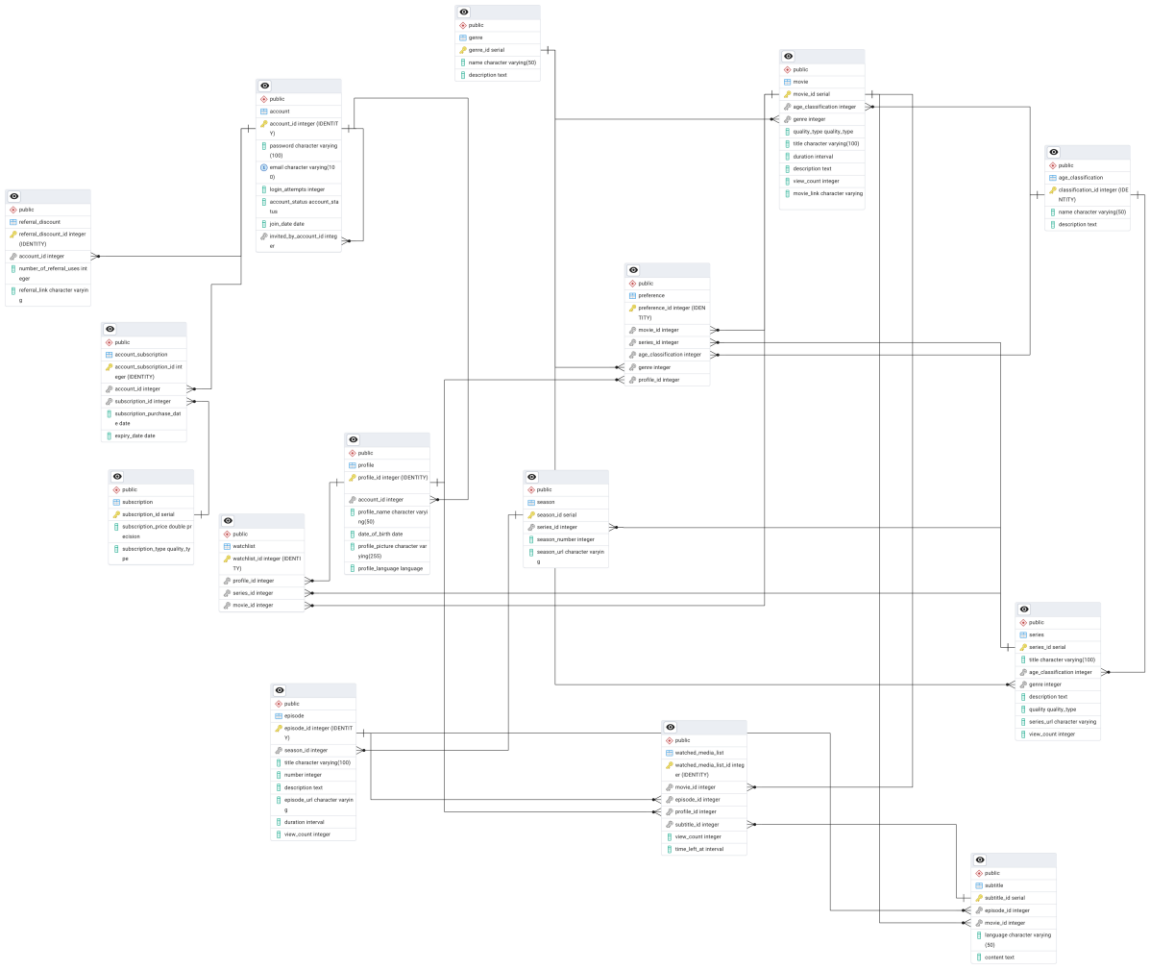


Cat image generated by The Cat API

The system design has been made to fit UML Specifications. Toy can find asta file for the class diagram here

DATABASE DESIGN

The database has been designed to accommodate the needs of all our entities. Below is the Database design:



The ERD of the system

View the asta file here

In addition to regular tables, we have several other tools to add a layer of abstraction, and to also make it easier to communicate with the database.

STORED FUNCTIONS

The choice of using stored functions for these SQL queries came from the fact that:

a. These queries are used very often
b. They do not do any other operations outside of SELECT statements so a stored procedure or a stored function would be inappropriate
c. The `api_role` has no access to the database directly, so they need to be able to get all this data through these stored functions.
d. Stored functions are more versatile than stored procedures because the allow you to carry out CRUD operations while still being able to return results

PLEASE NOTE: Because we have used stored functions in the API, you will likely see statements like SELECT * FROM public.update_account($1, $2, $3). This is PostgreSQL's way of calling stored functions as opposed to stored procedures which are called using CALL public.update_account($1, $2, $3).

There are 75 stored functions in the database so we will not enumerate them, but they take care of a wide range of CRUD operations including DELETEs, INSERTs, UPDATEs, and so on.

STORED PROCEDURES

Although stored functions are so versatile, we also use stored procedures. The list of stored procedures is:

- update_account(p_account_id, p_updated_account_id, p_updated_email, p_updated_account_status, p_join_date, p_status, p_email, p_password, p_account_status)
- public.delete_account_by_id(p_account_id, p_deleted_account_id, p_deleted_email, p_deleted_account_status, p_delete_date, p_status)
- public.create_account(p_password, p_email, p_invited_by_account_id, p_account_id, p_status)

The motivation behind these stored procedures is that they were made ACID compliant transactions and PostgreSQL does not support stored functions being ACID compliant.

VIEWS

Leaving out the view tables for the existing roles, we have 9 views in total:

- Account subscription details: information about all accounts that have subscription and the status of the subscription
- Active subscriptions: list of all active subscriptions
- All media: list of all movies, series, and episodes
- Episodes by seasons or series: list of all the episodes in a season or series (accepts id parameter)
- Movies by genre: list of movies in a genre
- Series by genre: list of series in a genre
- Profile watched media: list of watched movies and series for profile and time stamp (accepts id parameter)
- Referral stats: referral discounts and their statuses
- Series seasons: list of series in a season

TYPES

We have 3 enum types in the database:
account status:

- accepts values 'blocked', 'active', 'inactive'
- Is used by table account

Language:

- Accepts values 'English', 'Dutch', 'Pitjantjatjara'
- Is used by table profile

Quality type:

- Accepts values 'HD', 'UHD', 'SD'
- Is used by tables subscription, series, and movie

TRIGGERS

Deactivated Expired Accounts

- Function Name: deactivate_expired_accounts

- Trigger Action: This function checks if the expiry_date of an account has passed. If it has, it updates the account_status to 'inactive'.
- Purpose: Automatically deactivates accounts that have expired, ensuring that inactive accounts don't remain in the system with full access after their expiration date.
- How it works:
    - The trigger is executed when a record is inserted, updated, or deleted on the account table.
    - If the expiry_date is earlier than the current date, it sets the account_status to 'inactive'.

Enforce Profile Limits

- Function Name: enforce_profile_limit
- Trigger Action: This function ensures that an account does not exceed a limit of 4 profiles.
- Purpose: Prevents users from creating more than 4 profiles under the same account.
- How it works:
    - For an INSERT operation: It checks how many profiles are already associated with the account. If the count is 4 or more, it raises an exception and prevents the new profile from being created.
    - For an UPDATE operation: If the account_id is changing, the function checks if the new account has 4 or more profiles, raising an exception if necessary. If the account ID isn't changing, no check is made.
    - Exception Message: If the profile limit is reached, it raises an exception with a message indicating the account has already hit the maximum number of profiles.

Remove Watched Media from Watchlist

- Function Name: remove_watched_media_from_watchlist
- Trigger Action: This function removes a movie or episode from the user's watchlist once it has been watched.
- Purpose: Automatically removes media from the watchlist once it is marked as watched (either a movie or an episode).
- How it works:
    - If a movie_id is present, it deletes the entry from the watchlist associated with the profile and movie.

- If an episode_id is present, it deletes the entry from the watchlist for the associated series_id (by querying the episode table to find the series_id).
- This ensures that watched content is removed from the user's watchlist and keeps the data in sync.

Update Media Views

- Function Name: update_media_views
- Trigger Action: This function updates the view count for either a movie or episode whenever it is watched.
- Purpose: Tracks the number of times a movie or episode is viewed by incrementing the view_count each time it's watched.
- How it works:
  - If a movie_id is present in the new record, it increments the view_count of the movie in the movie table.
  - If an episode_id is present, it increments the view_count of the episode in the episode table.
  - This allows the system to keep track of how often movies and episodes are viewed, which could be useful for analytics and recommendations.

Update Referral Discount

- Function Name: update_referral_discount
- Trigger Action: This function updates the referral discount based on the status of the inviter and invitee accounts.
- Purpose: Ensures that referral discounts are updated properly depending on the status of the accounts involved in the referral.
- How it works:
  - It checks if both the invitee's (NEW.account_id) and the inviter's (NEW.invited_by_account_id) accounts are active.
  - If both accounts are active, it increments the number_of_referral_uses for the inviter's account, meaning they have successfully referred the invitee.
  - If either account is not active, it decrements the number_of_referral_uses, indicating the referral is no longer valid.
  - This function ensures that the referral system only rewards active accounts, maintaining the integrity of referral rewards.

ACID COMPLIANCE

Out of all procedures possible with the API, three procedures have been made ACID compliant. The create account function, the update account, and the delete account.

Account creation and deletion is an important and basic operation that affects many tables. Especially in the case of account deletion, we would have to delete all related profiles, then all related preferences to all related profiles. Because of this, ensuring that it is ACID-compliant helps maintain data integrity and ensures that no invalid, partial, or corrupt accounts are created in the system. Update account carries the same risk as create account, and atomicity is crucial for all three functions.

For these functions, we have set an isolation level of 'SERIALIZABLE' because it ensures that transactions are executed in such a way that the result is as if they were executed serially — one after another — without any overlapping. You may already be able to guess why this isolation level is a fit for these functions, but as they carry sensitive data AND a cascading series of queries, it is essential that the query is treated as isolated parts so that each one is fully finished and either passes or fails before the next one proceeds.

Please note: We chose to set the isolation level in the API end while ROLLBACKs were made in the database.

ADVICE ON DATABASE DESIGN

Optimization for Scaling and Performance

As the system scales, ensuring performance optimization is critical. One of the first areas to focus on is indexing. **Indexes** are necessary for improving query performance, especially for frequently queried columns such as account_id, profile_id, movie_id, and subscription_id. By creating indexes on foreign key columns, you can reduce the query time when searching for relationships between users, profiles, movies, and subscriptions.

Composite indexes can provide additional optimization. When certain fields are queried together often (e.g., profile_id, content_id, and watch_status in the watch_history table), creating composite indexes can significantly speed up these lookups.

Another key consideration for scaling the database is **partitioning**. For large tables, such as watch_history, movies, series, and subscriptions, partitioning them by fields like user_id or date can significantly improve performance and manageability. Partitioning tables based on time—such as partitioning watch_history by year or month—allows for faster access to recent data while facilitating the archiving of older records. This approach helps keep the database performance optimized without overwhelming it with excess data that doesn't need to be frequently queried.

Data Redundancy and Caching

In large-scale systems, data redundancy and caching play a crucial role in enhancing performance and reducing database load. One potential improvement is **denormalization**, where data is stored in a slightly redundant form to improve query efficiency, especially for reports or heavy queries. For example, creating materialized views to store precomputed results of frequently accessed data, like the number of views per movie, the average ratings for a movie/series, or the most popular genres, can significantly reduce the load on the database. These materialized views can be refreshed periodically to ensure that the data remains up to date while saving query time for users who access these reports regularly.

Another strategy is **caching**. Frequently accessed data can be cached either in-memory or in a distributed cache. By caching this data, the system can serve requests much faster, reducing the load on the database and improving the overall user experience.

DATABASE BACKUP ADVICE

In this document, we have gone into detail about how the database should be backed up and the things to consider during backup. There is also a sample .sql code here that can make the theorized concepts we have talked about much clearer.