Design Evidence Document

By Ameli Fernando & Polina Zueva

Table of Contents

Chapter 1: Introduction

This document details the design of a Netflix streaming platform, enabling users to sign up, manage profiles, watch content, and handle subscriptions. The architecture also incorporates specialized roles (Admin, Mediator, Junior Staff) for content management.

Chapter 2: Description of the system

The developed application is a **Netflix Clone backend system** that provides user management, content cataloging, subscription handling, and secure access control. It is designed to simulate the essential functionalities of a streaming platform while following best practices for database management, API development, and security.

The system follows a modern, layered 3-tier architecture:

1. **Front-end (Presentation Layer):**
   - **A** React.js web application providing a management panel for Netflix employees.
   - Handles data visualization, user input, and authentication workflows**.**
   - Communicates with the backend API via secure HTTP (RESTful) requests**.**

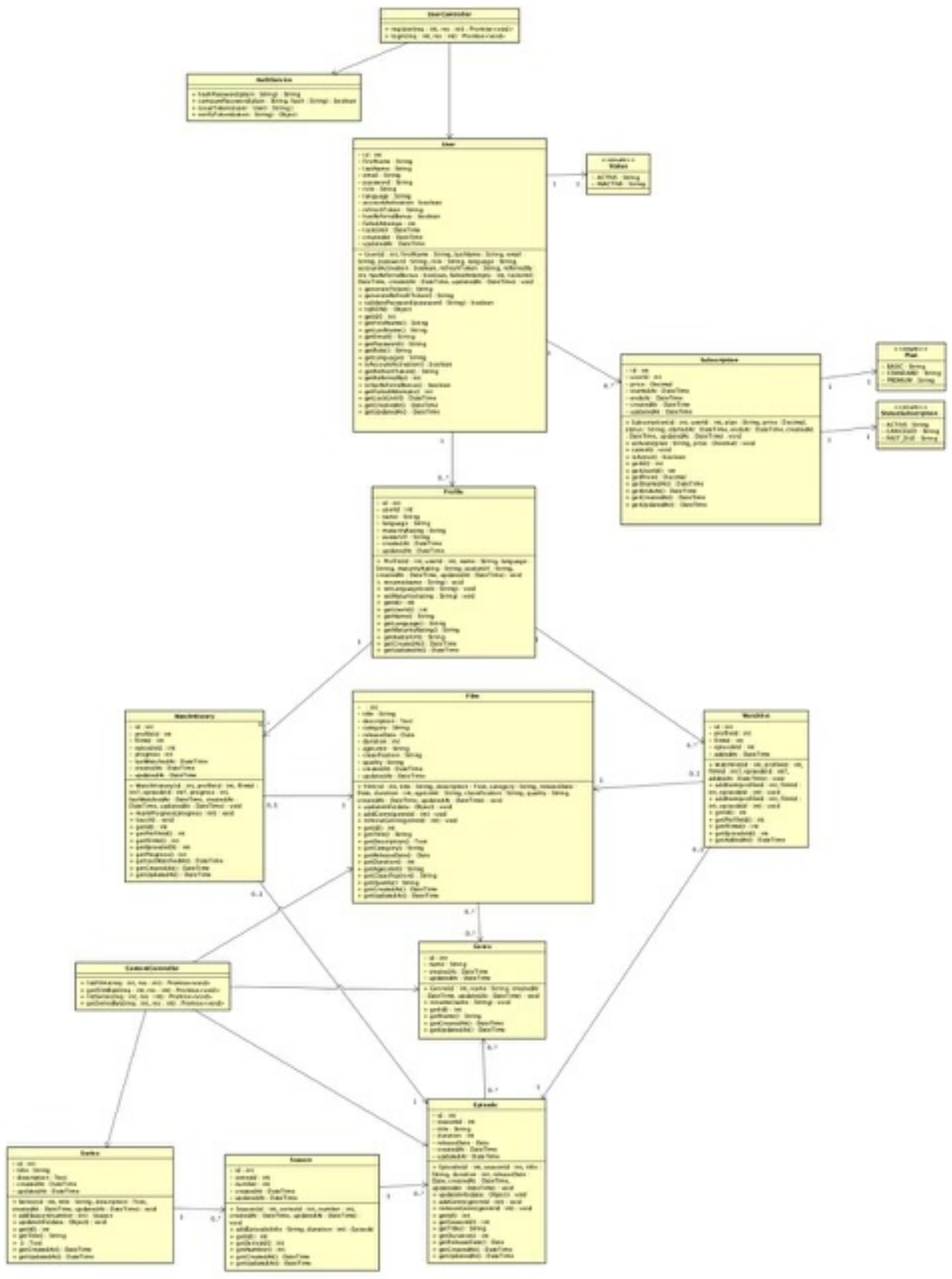
2. **API (Application Layer):**
   - Built with Node.js and Express, serving as the business logic layer.
   - Manages user authentication and authorization using JWT tokens.
   - Implements CRUD operations for films, series, users, and subscriptions.
   - Enforces validation, error handling, and security checks before interacting with the database.
   - Uses Sequelize ORM for structured database interactions, migrations, and seeding.

3. **Database (Data Layer):**
   - A PostgreSQL relational database system storing all application data (users, films, series, seasons, episodes, and subscriptions).
   - Maintains referential integrity using foreign keys and cascading rules.
   - Uses Views and Stored Procedures to optimize queries and encapsulate database logic.
   - Employs Triggers (e.g., auto-updating updatedAt columns) and **indexes** (e.g., partial index on active subscriptions) to maintain consistency and improve performance.

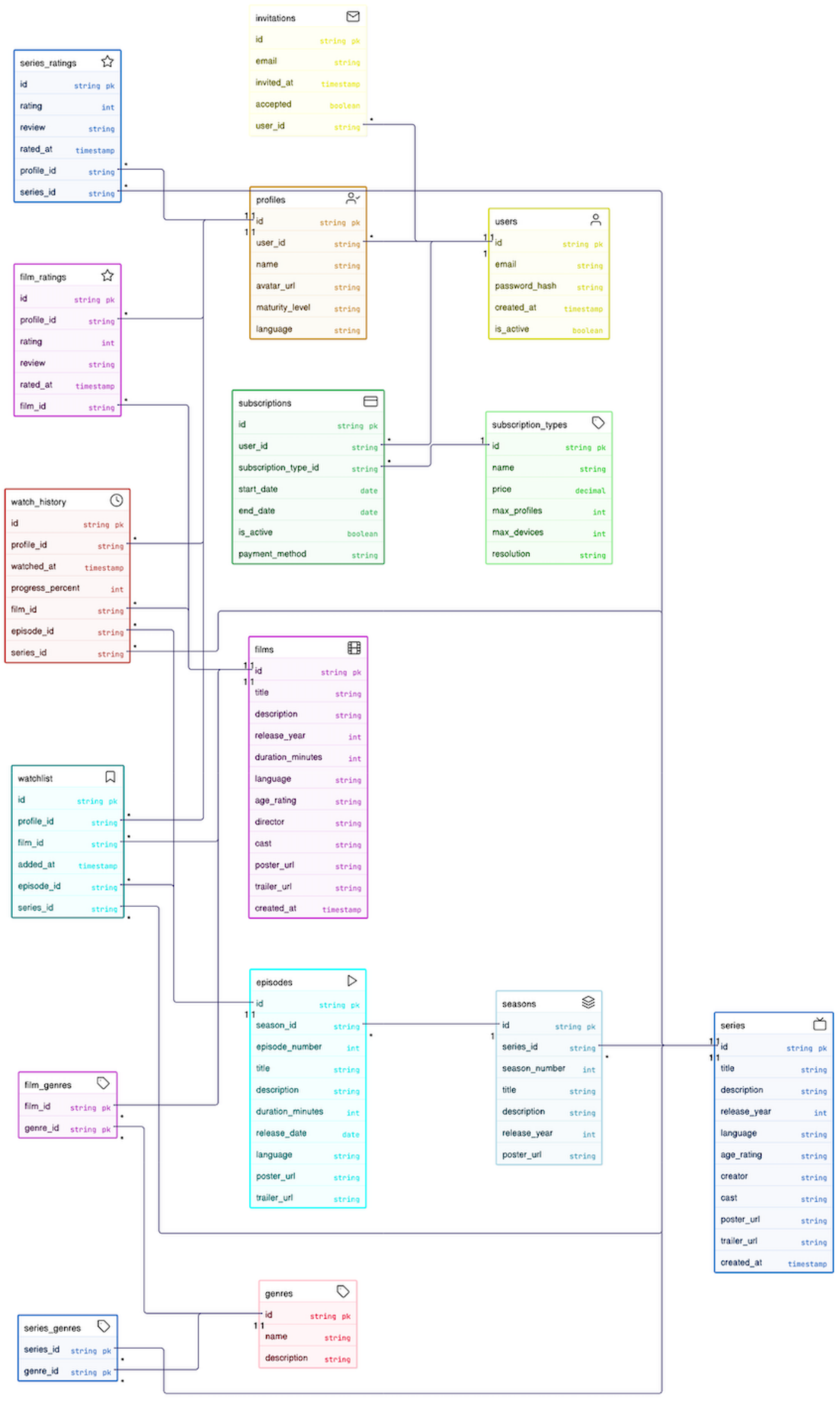
Authorization & Users

User Role	Database Permissions (GRANT)
Senior	SELECT, INSERT, UPDATE, DELETE on all tables. EXECUTE on all procedures. Can access all views.
Medior	SELECT on all tables *except* financial ones. SELECT on view_user_watch_history. EXECUTE on non-financial procedures.
Junior	SELECT on non-financial, non-PII tables. SELECT on view_profiles_anonymous and view_user_watch_history. EXECUTE on a limited set of procedures.
API	**EXECUTE** on all Stored Procedures. **SELECT** on all Views. **DENY** SELECT, INSERT, UPDATE, DELETE **ON ALL TABLES**. This forces the API to use the abstracted interface.

Chapter 3: Class Diagram



Chapter 4: Entity relationship diagram



Chapter 5: Testing

- **Unit Test** – Focuses on validating the smallest building blocks of the application independently, such as verifying password hashing or ensuring token generation behaves correctly.
- **Authorization Test** – Ensures that secured API routes are only accessible with a valid JWT, rejecting requests from unauthenticated or invalid users.
- **Functional Test** – Examines how endpoints respond to both correct and incorrect inputs, checking that the returned status codes and responses match expectations.
- **Security Test** – Confirms the system does not expose sensitive details (like plain-text passwords or raw refresh tokens) in its API outputs, adhering to secure practices.
- **Error Handling Test** – Checks that when unexpected issues occur (for example, database connection errors), the system responds with clear, safe error messages instead of exposing stack traces or crashing.

Unit

Test Case ID	Test Type	Description	Preconditions	Steps	Expected Results
TC-unit-1	Unit	Verify token contains correct claims & expiry	User model with generateToken() method	User model with generateToken() method	JWT payload has id, email, and valid expiry

Test Case ID	Test Type	Description	Preconditions	Steps	Expected Results
TC-unit-2	Unit	Verify password hashing & validation	Verify password hashing & validation	Compare password with correct & incorrect inputs	Returns true for correct password, false otherwise

Authorization

Test Case ID	Test Type	Description	Preconditions	Steps	Expected Results
TC-auth-01	Authorization	Protected route requires valid JWT	API running with /api/profile protected	Request without token. 2) Request with valid token.	Returns 401/403 without token, 200 with token

Functional

Test Case ID	Test Type	Description	Preconditions	Steps	Expected Results

TC-func-01	Functional	Fetch non-existent user returns error	Users table does not contain ID 999999	Send GET /api/users/999999	Returns 404 or 400 with message

Test Case ID	Test Type	Description	Preconditions	Steps	Expected Results
TC-func-02	Functional	Registration successfully creates a user	API running, DB empty or unique email	Send valid registration request	Returns 201 with created user JSON (sans password)

Security

Test Case ID	Test Type	Description	Preconditions	Steps	Expected Results
TC-sec-01	Security	Registration response does not expose sensitive data	API running, valid registration body	Send POST /api/users/register with email & password	Send POST /api/users/register with email & password

Error Handling

Test Case ID	Test Type	Description	Preconditions	Steps	Expected Results
TC-err-01	Error Handling	System returns safe error for internal failure/db failure	Controller modified to simulate DB error	Trigger login with header X-Inject-Failure: true	Returns 500 with generic safe message, no stack trace

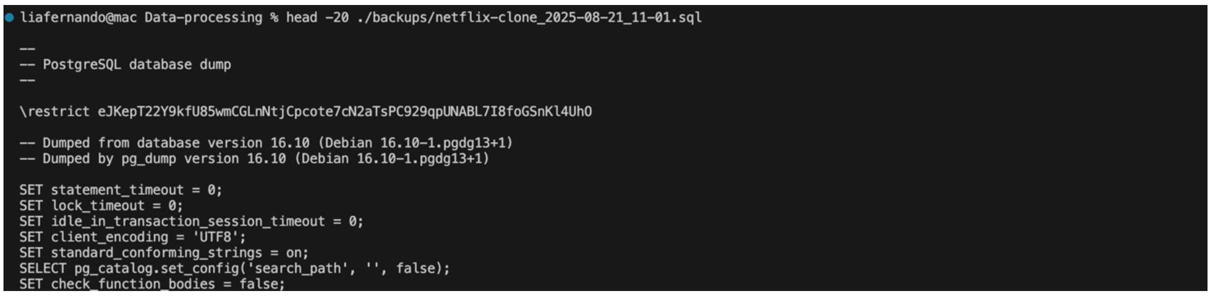
Chapter 6: Backup Strategy

The database backup strategy for this project is designed to guarantee data safety, reliability, and recoverability in case of failure or data loss. Backups of the PostgreSQL database are created automatically every day using the pg_dump utility, with each dump stored in a compressed format to save space.

Backup Retention Policy

- Daily backups are retained for 7 days.
- Weekly backups are retained for 1 month.
- Monthly backups are retained for 6 months.
- Older backups are pruned automatically to save storage space.

To maintain confidence in the reliability of the backup process, regular restore tests are performed. These tests involve restoring a backup to a clean PostgreSQL instance and verifying that the data and schema can be successfully recovered. Alongside the backup process, the database schema itself is version-controlled through Sequelize migrations, which allows the structure to be reconstructed at any time. This combination of automated backups, controlled retention, off-site storage, and schema versioning ensures that the system can quickly recover from unexpected failures while minimizing the risk of permanent data loss.



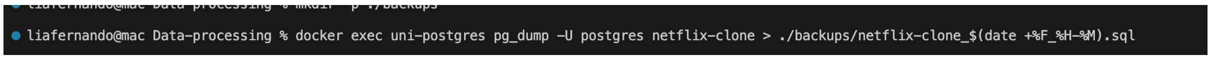
```
● liafernando@mac Data-processing % head -20 ./backups/netflix-clone_2025-08-21_11-01.sql

  --
  -- PostgreSQL database dump
  --

  \restrict eJKepT22Y9kfU85wmCGLnNtjCpcote7cN2aTsPC929qpUNABL7I8foGSnKl4UhO

  -- Dumped from database version 16.10 (Debian 16.10-1.pgdg13+1)
  -- Dumped by pg_dump version 16.10 (Debian 16.10-1.pgdg13+1)

  SET statement_timeout = 0;
  SET lock_timeout = 0;
  SET idle_in_transaction_session_timeout = 0;
  SET client_encoding = 'UTF8';
  SET standard_conforming_strings = on;
  SELECT pg_catalog.set_config('search_path', '', false);
  SET check_function_bodies = false;
```



```
● liafernando@mac Data-processing % mkdir -p ./backups

● liafernando@mac Data-processing % docker exec uni-postgres pg_dump -U postgres netflix-clone > ./backups/netflix-clone_$(date +%F_%H-%M).sql
```

Screenshot showing the PostgreSQL backup process with pg_dump, where the database dump is generated inside the Docker container and stored with a timestamped filename for reliable recovery