

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа

Выполнил:

Скирляк Ярослав, К3439

Проверил:

Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

Необходимо спроектировать базу данных для микросервисного сервиса совместного подбора фильмов (по аналогии с сервисом «Что?»), с использованием PostgreSQL для хранения основной информации и Redis – для хранения текущих сессий подбора фильмов. База данных должна удовлетворять следующим требованиям:

1. **Аутентификация и пользователи:** Нужно хранить информацию о пользователях: никнейм, email и пароль (в базе хранится хеш пароля). Реализовать возможность двухфакторной аутентификации через Google.
2. **Друзья:** Должна быть возможность отправлять пользователям заявки в друзья и подтверждать дружбу. Необходимо хранить список подтвержденных друзей. При этом история взаимодействий между друзьями (например, сообщения) не сохраняется.
3. **Комнаты (сессии подбора):** Для каждой сессии совместного подбора фильмов создаётся отдельная "комната". Комната существует только на время сеанса и закрывается после его завершения.
4. **Мэтчинг фильмов:** Предусмотрен функционал свайпов по фильмам – пользователи отмечают фильмы «лайком» или «дизлайком». При этом сохраняются только успешные совпадения (мэтчи) – то есть фильмы, которые понравились **обоим** пользователям. Информация о каждом отдельном свайпе (лайк/дизлайк на фильм) не сохраняется.
5. **Фильмы:** Информация о фильмах берётся из внешнего API (например, TMDb API). Нет необходимости кэшировать данные о фильмах в PostgreSQL (т.е. не требуется создавать отдельную таблицу фильмов в нашей базе).
6. **Redis:** Используется для хранения данных текущей комнаты/сессии подбора фильмов (в памяти). Постоянные данные о пользователях, друзьях и совпадениях хранятся в PostgreSQL.

Ход работы

1. Аутентификация и пользователи

Для хранения данных о пользователях создана таблица **Users**. В ней содержатся основные поля профиля и данные для аутентификации:

- **nickname** – уникальный никнейм пользователя (строка).
Используется как отображаемое имя в сервисе.
- **email** – электронная почта (строка), уникальный идентификатор для входа в систему.
- **password_hash** – хеш пароля пользователя (строка). При регистрации/изменении пароля пароль хешируется (например, алгоритмом bcrypt) и сохраняется.
- **google_2fa_secret** – секретный ключ для двухфакторной аутентификации через Google Authenticator (строка). Этот ключ генерируется для каждого пользователя при подключении 2FA и используется для проверки одноразовых кодов. Поле может быть NULL, если пользователь не подключил 2FA.
- **is_2fa_enabled** – флаг (BOOLEAN), указывающий, включена ли у пользователя двухфакторная аутентификация. Значение **TRUE** означает, что 2FA настроена и требуется при входе, **FALSE** – 2FA не используется.

2. Друзья (добавление в друзья)

Для реализации функционала друзей создана таблица **Friendships** (дружеские связи). Она хранит отношения между пользователями, включая отправленные заявки и подтвержденную дружбу.

Структура **Friendships**:

- **user_id1** – идентификатор первого пользователя в дружбе (внешний ключ на Users.id).
- **user_id2** – идентификатор второго пользователя (внешний ключ на Users.id).

- **status** – статус отношений: например, **"pending"** (заявка отправлена, ожидает подтверждения) или **"accepted"** (дружба подтверждена).
- **created_at** – дата и время создания записи (отправки заявки в друзья), проставляется по умолчанию текущим временем.

3. Комнаты (сессии подбора фильмов)

Совместный подбор фильма осуществляется в рамках временной сессии, называемой "комната". Комната создаётся, когда пользователи начинают совместный поиск фильма, и существует только пока идёт этот процесс. Согласно требованиям, данные комнаты **не сохраняются в PostgreSQL**, а хранятся во временном хранилище Redis.

- список участников (идентификаторы двух пользователей, которые находятся в данной комнате),
- текущий показываемый фильм или индекс текущего фильма в подборке,
- отметки «нравится/не нравится» от каждого участника по текущему фильму,
- другие временные данные, необходимые для логики (например, сколько фильмов пролистали, найден ли мэтч).

Когда сессия завершается (либо пользователи нашли общий фильм, либо прервали подбор), соответствующие данные в Redis удаляются. Таким образом, **комнаты являются эфемерными** и после завершения сеанса **никакая информация о процессе подбора (кроме результата мэтча) в основной базе не остаётся**.

4. Мэтчинг фильмов (механика свайпов)

В процессе совместного просмотра подборки фильмов каждый пользователь выражает своё мнение: «лайк» (нравится, потенциально хочу смотреть) или «дизлайк» (неинтересно) на предлагаемые системой фильмы. Эта механика сходна с приложениями знакомств: показывается один и тот же фильм обоим пользователям, и они независимо свайпают влево/вправо.

Сохранение совпадений (matches): Если оба пользователя поставили лайк одному и тому же фильму – это считается успешным совпадением (мэтч). Такой результат фиксируется в базе данных. Создана таблица **Movie_Matches** для хранения всех найденных парных совпадений по фильмам.

Структура **Movie_Matches**:

- **user_id1, user_id2** – идентификаторы двух пользователей, между которыми произошло совпадение (оба поставили лайк данному фильму). Это внешние ключи, ссылающиеся на таблицу **Users**. Аналогично дружбе, принимаем convention **user_id1 < user_id2** для упорядочения и вводим **CHECK** на неравенство, чтобы не было дубликатов в обратном порядке.
- **movie_id** – идентификатор фильма, на котором сошлись оба пользователя. Поскольку данные о фильмах берутся из API TMDb, здесь хранится, например, TMDb ID фильма (целое число). Этот идентификатор позволяет при необходимости получить подробную информацию о фильме через внешнее API. В нашем хранилище он служит ссылкой на фильм.
- **matched_at** – отметка времени, когда мэтч был зафиксирован (TIMESTAMP). Позволяет хранить историю совпадений и, например, отображать пользователям, когда именно они нашли тот или иной фильм.

Первичный ключ для **Movie_Matches** можно определить как составной по трём полям (**user_id1, user_id2, movie_id**), то есть одно и то же сочетание двух пользователей и одного фильма сохранится только один раз. Таким образом исключаются повторные записи о совпадении на одном и том же фильме.

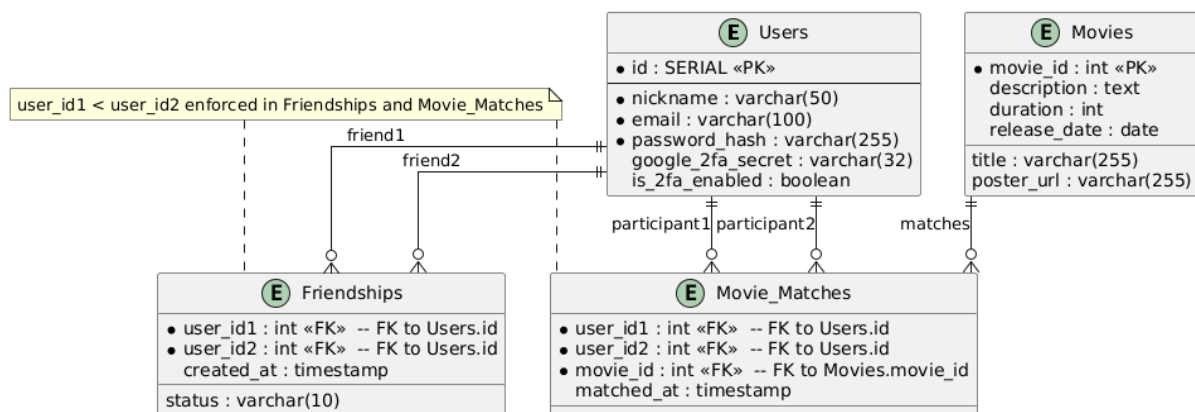
5. Фильмы и интеграция с TMDb

Согласно требованиям, информация о фильмах берётся через TMDb API по запросу в реальном времени, и в **PostgreSQL** не создаётся постоянной таблицы фильмов. Это значит, что мы не кэшируем подробные данные (название, описание, постер и т.д.) внутри нашей базы.

6. Использование Redis для сессий

Как отмечалось, Redis используется для хранения данных текущей сессии подбора. Во время активной комнаты в Redis может храниться структура примерно следующего вида (псевдо-структура ключей):

- `room:{room_id}:users` – список или множество участников (IDs пользователей).
- `room:{room_id}:current` – ID текущего фильма, который предлагается обоим пользователям.
- `room:{room_id}:likes` – отметки лайков (например, хэш, где ключи – `user_id`, значения – лайк/дизлайк/не проголосовал). Либо отдельные ключи `room:{room_id}:user:{user_id}:liked = true/false`.
- Возможно, дополнительные ключи, например `room:{room_id}:status` (статус сессии, найден ли мэтч), или очередь/список оставшихся фильмов для показа.



```
@startuml
skinparam linetype ortho

entity "Users" as U {
    *id : SERIAL <<PK>>

```

```
--

*nickname : varchar(50)

*email : varchar(100)

*password_hash : varchar(255)

google_2fa_secret : varchar(32)

is_2fa_enabled : boolean
}

entity "Friendships" as F {

    *user_id1 : int <<FK>> -- FK to Users.id

    *user_id2 : int <<FK>> -- FK to Users.id

    status : varchar(10)

    created_at : timestamp
}

entity "Movie_Matches" as M {

    *user_id1 : int <<FK>> -- FK to Users.id

    *user_id2 : int <<FK>> -- FK to Users.id

    *movie_id : int <<FK>> -- FK to Movies.movie_id

    matched_at : timestamp
}

entity "Movies" as MOV {
```

```

*movie_id : int <<PK>>

title : varchar(255)

description : text

poster_url : varchar(255)

duration : int

release_date : date
}

note "user_id1 < user_id2 enforced in Friendships and
Movie_Matches" as N1

N1 .. F

N1 .. M

U ||--o{ F : friend1
U ||--o{ F : friend2
U ||--o{ M : participant1
U ||--o{ M : participant2

MOV ||--o{ M : "matches"

@enduml

```

SQL-таблица:

```

CREATE TABLE users (

    id                SERIAL PRIMARY KEY,

```



```

    nickname          VARCHAR(50)  NOT NULL UNIQUE,

    email              VARCHAR(100) NOT NULL UNIQUE,

    password_hash      VARCHAR(255) NOT NULL,

    google_2fa_secret  VARCHAR(32),

    is_2fa_enabled     BOOLEAN NOT NULL DEFAULT FALSE
);

CREATE TABLE friendships (

    user_id1          INT NOT NULL,

    user_id2          INT NOT NULL,

    status            VARCHAR(10) NOT NULL DEFAULT 'pending',

    created_at        TIMESTAMPTZ NOT NULL DEFAULT NOW(),

    PRIMARY KEY (user_id1, user_id2),

    FOREIGN KEY (user_id1) REFERENCES users(id) ON DELETE CASCADE,

    FOREIGN KEY (user_id2) REFERENCES users(id) ON DELETE CASCADE,

    CHECK (user_id1 <> user_id2),

    CHECK (user_id1 < user_id2)
);

CREATE TABLE movies (

    movie_id          INT PRIMARY KEY,

    title             VARCHAR(255) NOT NULL,

    description        TEXT,

    poster_url        VARCHAR(255),

    duration           INT,

    release_date       DATE

```

```
);

CREATE TABLE movie_matches (

    user_id1    INT NOT NULL,

    user_id2    INT NOT NULL,

    movie_id    INT NOT NULL,

    matched_at  TIMESTAMPTZ NOT NULL DEFAULT NOW(),

    PRIMARY KEY (user_id1, user_id2, movie_id),

    FOREIGN KEY (user_id1) REFERENCES users(id) ON DELETE CASCADE,

    FOREIGN KEY (user_id2) REFERENCES users(id) ON DELETE CASCADE,

    FOREIGN KEY (movie_id) REFERENCES movies(movie_id) ON DELETE CASCADE,

    CHECK (user_id1 <> user_id2),

    CHECK (user_id1 < user_id2)

);
```

Вывод

В ходе работы спроектирована и реализована схема базы данных для сервиса совместного выбора фильмов, отвечающая всем заданным требованиям.