Software documentation

DB Scheme structure:

Person(ID, name)

Director(ID, gender, birthday, bio)

Actor(ID, gender, birthday, bio, popularity)

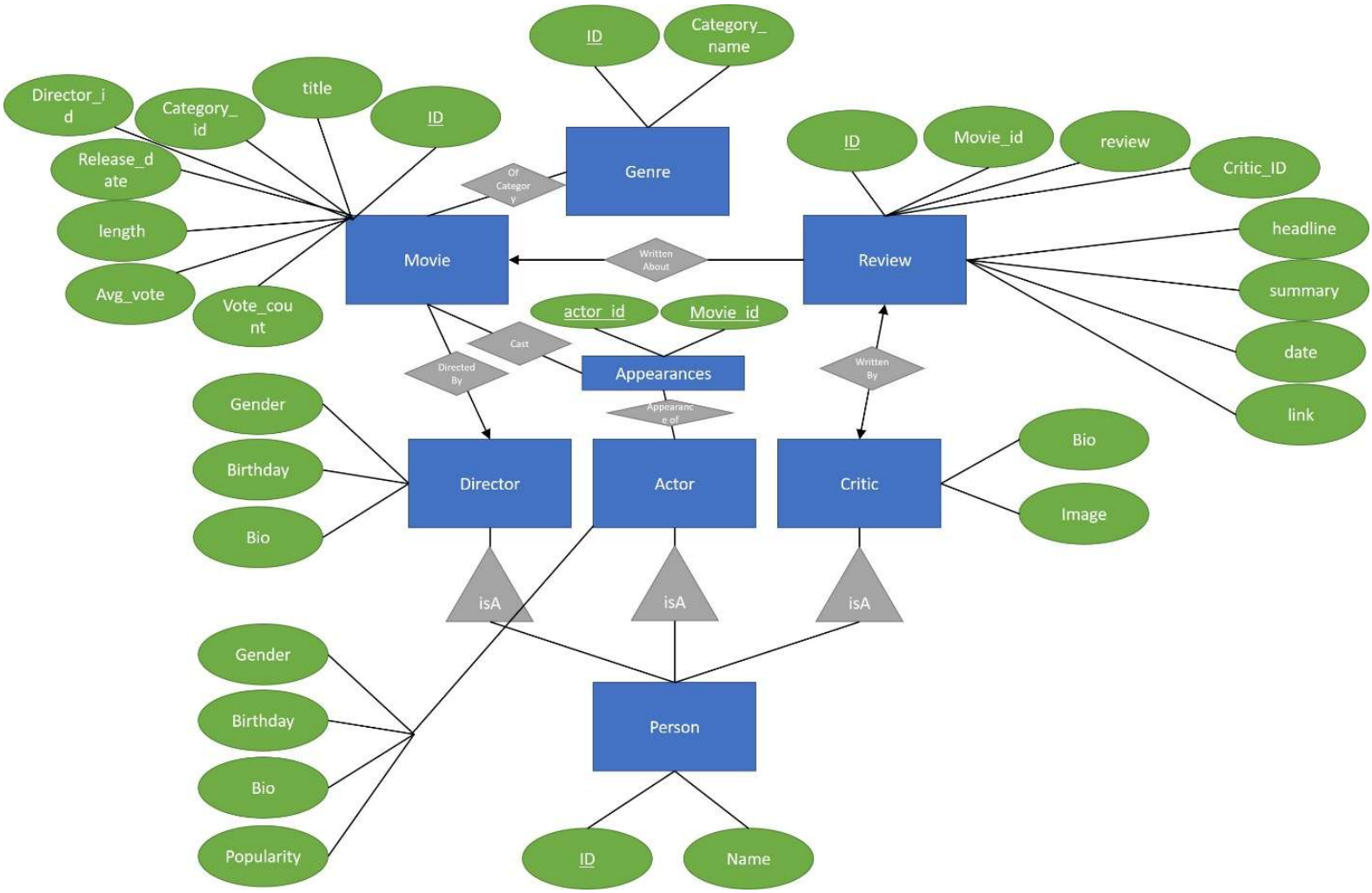
Critic(ID, bio, image)

Review(ID, movie_id, critics_pick, critic_id, headline, summary, date, link)

Movie(ID, title, category_id, director_id, release_date, length, vote_avg, vote_count)

Appearances(movie_id, actor_id)

Genre(ID, category_name)



DB design explanation:

We created a database for movie reviews, and we decided we need to represent the critics, the reviews, the movies and the relations between them.

We decided that we want to exam the relations between reviews and movies from serval aspects of the movie, such as the genre of the movie, the director and the actors that appeared in the movie.

Hence, we decided to have a Person table that represent all the people that are involved in the DB, and three different tables that inherits it (Director, Actor, Critic).

We also figured we need to have a relation table for actors and movies because it is a many-to-many relation, each actor can appear in many movies, and in each movie there are many actors. Therefore, we created the Appearances table that saves space in the DB representing that exact movie-actor relation.

We decided to take only relevant information from the APIs we used and store them into the DB tables attributes.

We think this the design is the most efficient design possible for our application needs because it doesn't waste storage space with tables that have too many attributes and in the same time it doesn't require too many tables to maintain.

In the beginning we thought to not include the Person table and then decided we need it because we figured there are actors that are also directors and then if they appear in both tables we waste space of writing their names twice.

DB optimization performed:

To optimize the use of our DB we used several primary keys and foreign keys to ensure queries are performed as fast as possible. Please note that every table that starts with Django_ or auth_ was automatically generated by Django for internal purposes. Our data populates the tables specified in the schema.

Primary keys (Table-key):

Actor-ID
Appearances-movie_id, actor_id
Critic-ID
Director-ID
Genre-ID
Movie-ID
Person-ID
Review-ID

Foreign keys (Table-key-referenced table)
Actor-ID-Person
Appearances-actor_id-Actor
Appearances-movie_id-Movie
Critic-ID-Person
Director-ID-Person
Movie-category_id-Genre
Movie-director_id-Director
Review-critic_id-Critic
Review-movie_id-Movie
Full text(Table-(columns)):
Review-(summary,headline)

Queries

1. query = "select Movie.title as Movie_title, Person.name as critic, summary, link from Person join Review on" \
   " Person.ID = Review.critic_id join Movie on Movie.ID = Review.movie_id WHERE Person.name LIKE %s"
   Gets the review of a critics with the given term in their name.
   Uses the FK relations between Critic-Person, Movie-Critic and Review-Movie to optimize the query. Supported by the structure of the DB using those FKs.
2. query = "SELECT Movie.title, Movie.vote_avg, summary, link FROM Review JOIN Movie ON Movie.ID = Review.movie_id" \
   " order by Movie.vote_avg desc"
   Retrieves the user ratings of the movies with their vote average and other movie\review details.
   Uses the FK relations between Movie-Review.
3. query = "SELECT Movie.title AS Movie_title, summary, link FROM Review JOIN Movie ON Movie.ID = Review.movie_id" \
   " WHERE critics_pick=1"
   Gets the movies and reviews that critics have recommended.
   Critics_pick=1 is our way of indicating that a critic has recommended a movie.
   Uses the FK relations between Review-Movie
4. query = "SELECT Movie.title AS Movie_title, summary, link FROM Review JOIN Movie ON Movie.ID = Review.movie_id" \
   " WHERE MATCH(summary, headline) against (%s IN NATURAL LANGUAGE MODE) "
   Gets all movies and reviews that have a similar phrase to the given term.
   Full text query supported by our DB indices.
   Uses FK relations between Movie-Review

5. query = "SELECT name, amount, picks, bio, image from (SELECT Person.name, COUNT(*) as amount, SUM(critics_pick)" \
   " as picks, Person.ID FROM Person JOIN Review ON Person.ID = Review.critic_id JOIN Critic ON Critic.ID" \
   " = Person.ID WHERE Person.name = %s GROUP BY critic_id) as T JOIN Critic on T.ID = Critic.ID"
   Retrieves the number of movies a certain critic recommended out of all the movies they have reviewed with additional information about the critic.
   Our DB supports this by setting critics_pick=0 for unrecommended movies and 1 for recommended movies.
6. query = "SELECT Movie.title AS Movie_title, summary, link FROM Review JOIN Movie ON Movie.ID = Review.movie_id " \
   "JOIN Genre on Genre.ID = Movie.category_id WHERE critics_pick = 1 AND category_name=%s"
   Gets the movies that critics recommended for a specific genre.
   Supported by the FK relations between Movie-Genre, Movie-Review
7. query = "SELECT AVG(Movie.vote_avg) as vote, AVG(Movie.vote_count), category_name FROM Movie join Genre ON Movie.category_id = Genre.ID group by Movie.category_id order by vote desc"
   Gets the average user score of movie genres.
   Supported by the column types and the FK relation Movie-Genre

8. query = "SELECT Person.name, Movie.title, Review.link, Review.summary, Actor.popularity FROM Review JOIN Movie on Movie.ID = Review.movie_id JOIN Appearances on Movie.ID = Appearances.movie_id JOIN Actor on Appearances.actor_id = Actor.ID Join Person on Person.ID = Actor.ID WHERE Person.name LIKE %s ORDER BY Actor.popularity DESC"
   Retrieves reviews on movies that a certain actor took part in along with some information about the actor.

9. query = "SELECT Person.name, Director.birthday, Director.bio, Movie.title, Review.link, Review.summary from Director JOIN Movie on Movie.director_id = Director.ID Join Review on Movie.ID = Review.movie_id Join Person on Person.ID = Director.ID WHERE Person.name LIKE %s"
   Retrieves reviews for movies of a certain director along with some information on that director

Code Structure

API

Api_creds.json- configuration file containing relevant API and DB keys
get_and_upload.py- main script, uses the API modules in order to retrieve the data and upload it to the DB
new_york_api.py- wrapper module for NYT's API
tmdb.py- wrapper module for TMDB's API

Application (Django framework)

__init__.py – used for the machine to recognize directories as python source.
el_critico (directory):

- Asgi.py, wsgi.py- general configurations used by Django on different engines
- Settings.py- Django configuration, includes the debug, templates, static files, DB and installed apps configuration
- Urls.py- application urls, includes built-in admin support (unused in our project)

Finder (directory):

- Migrations- Used by Django to recognize DB
- Static- contains our static files (Images and CSS files)
- Templates- Contains our HTML files
- Admin.py – unused, auto generated by Django
- Apps.py – apps details for Django to recognize it
- Models.py – Data base model for each table, used "python manage.py inspectdb" to generate it which reads the data from our DB and generates models accordingly.
- Tests.py – unused, auto generated by Django.
- Urls.py – url mapping relative to "/finder". Each mapping points to a view class.
- Views.py – Each class in views contains a "get" method for GET requests and blocks any other request type. View per Query and for errors 500 and 404.

Manage.py- Django helper script to run all the different Django utilities

API description

We used 2 APIs, TMDB: https://www.themoviedb.org/ and New York Times: https://developer.nytimes.com/docs/movie-reviews-api/1/overview
TMDB

A free to use API that contains plenty of information about movies and user scores.
We used this API to get our Movies details including the director and actors.

Used Endpoints:

- /genre/movie/list
- /movie/top_rated
- /movie/popular
- /movie/{movie_id}/credits
- /person/{person_id}
- /movie/{movie_id}

New York Times

A free to use (under limitations) API that contains all of the New York Times movies reviews along with the Critics details

Used Endpoints:

- /critics/{reviewer}.json
- /reviews/all.json

External Libraries Used

- Django, version 3.1.2
- Mysqlclient, version 1.4

General Flow of Application

- Read user manual to deploy and start the application (or used our deployed application on http://delta-tomcat-vm.cs.tau.ac.il:40996/ if it is not deleted)
- When a request comes in, the application will attempt to map it to the relevant view function. If no such mapping exists a 404-erorr page will be displayed.
- When a request is mapped the view will check its type, GET requests will be mapped to get(request) function of that view while all other types of requests will cause a 403 error.
- Get(request) will retrieve the relevant data from the DB and render an HTML with that data. It will then display that data on the browser (or return the HTML code if using a script like CURL to call that endpoint).
- If a failure happens at any time a 500-error page will be displayed.