Design Document

I. Project Overview - Diego and VC

In a world where streaming services offer endless movie options, it can be overwhelming to find the perfect film to watch. That is why we developed MovieDB, a personalized movie recommendation application aimed at simplifying the movie selection process. Our platform uses advanced algorithms that process user data, including viewing history movie ratings, movie tags, and user preferences to generate customized movie suggestions.

Our goal is to provide users with an enjoyable movie watching experience by presenting them with movies that align with their interests and preferences. The MovieDB is intuitive and user-friendly, making it effortless for movie enthusiasts of all ages to discover new movies that they would like to watch. Additionally, the movie providers who offer streaming services benefit from our system, as it directs users to their platforms, increasing their viewership.

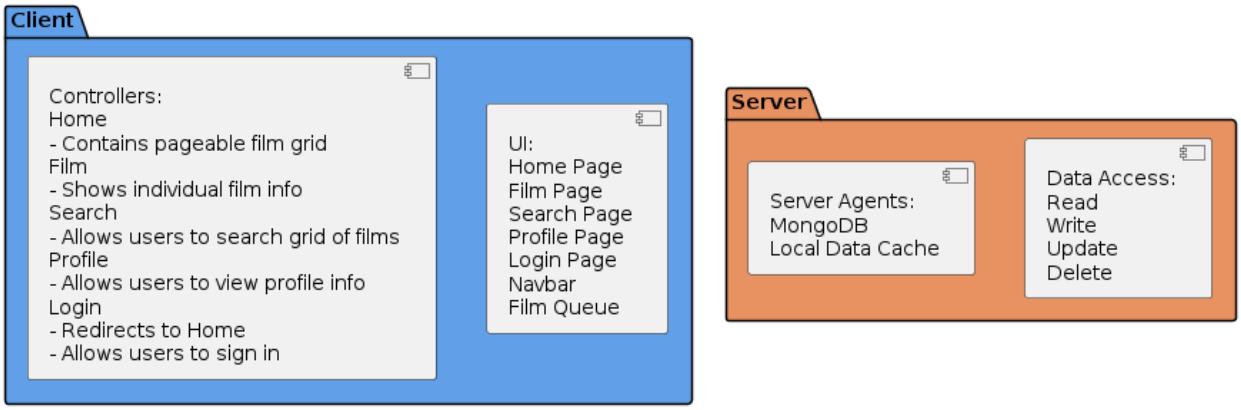
With MovieDB, movie enthusiasts can say goodbye to tedious and time-consuming movie searches that sometimes last longer than the movie itself. Our platform simplifies the movie selection process, allowing the user to spend less time searching and more time enjoying their favorite movie.

II. Architectural Overview - Will

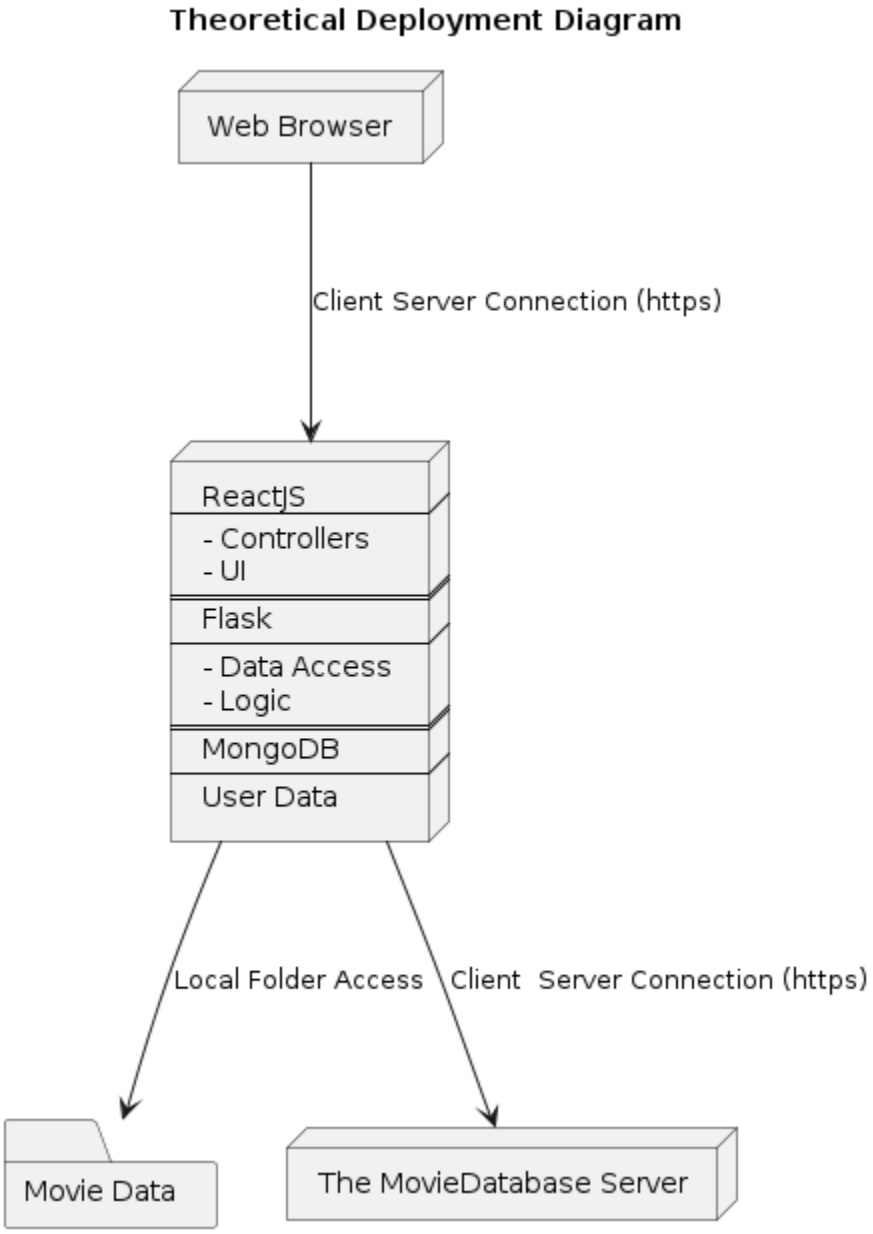
MovieDB is composed of three layers: Frontend, Backend, and the Database. To start, the frontend is made using ReactJS, a Javascript framework that allows the programmer to construct a site using modular components. The backend is made using Flask, a simple web framework that is built on Python. Our chosen database is MongoDB, which

we use to store user information. We also have individual data stores in CSV files that contain information about films. This setup was chosen because everyone in the project was most familiar with React as well as Python, making it an easy choice. We did not consider any alternatives at any point, since it seemed unimportant compared to the ease of use provided by the current setup. Besides familiarity with the architecture, the choice to use a ReactJS + Python setup is really useful for projects that require machine learning and AI. Our project contains minimal AI features, but we do make use of the sklearn library in Python, and at points throughout the project considered using more NLP based libraries (such as Spacy). Because of this, we decided to use Python and Flask. As for the frontend, ReactJS is one of the most popular frameworks, so it is logical to use it during a project that attempts to emulate a real world setting.

A. Subsystem Architecture - Will



III. Deployment Architecture



This is how the current deployment architecture looks. It works about as expected for a simple Flask / React application, the only real quirk is the inclusion of The MovieDatabase Server, which we have no control over but rely on in order to get posters. The inclusion of that server may be removed if we are able to store the posters locally, in which case it's functionality will be taken by the Movie Data folder.

A. Persistent Storage - VC

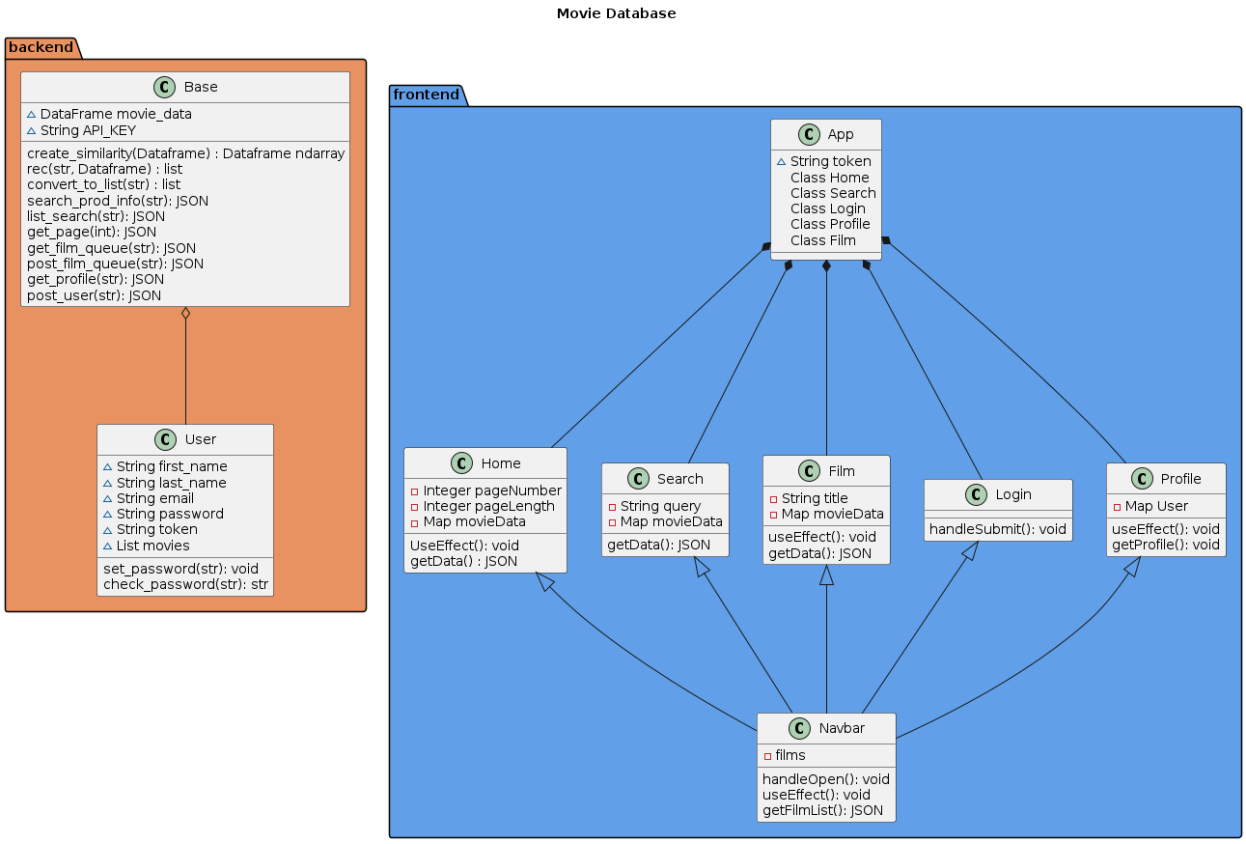
We stored our information in a database using MongoDB. The MongoDB database **user** model that includes first_name, last_name, email, password_hash has a list of their selected movies called movie, and a token and a session_id field. The user's email address is required and must be unique. The database is accessed using a Python client connected to the user ***"users"*** collection. The user model includes methods to set and check the password hash using the bcrypt library. The "movie" field is a list of strings that stores the user's selected movies. The "token" field is a string that stores the user's authentication token, which is used to authenticate the user for future requests. The "session_id" field is a string that stores the user's session ID, which is used to maintain the user's session state between the requests. The database allows for efficient lookup of users by email through indexing. To create a new user, insert a new document in the users collection with the required fields. To retrieve a user, use a query to find the document with the matching email field. To update a user's movie list, retrieve the user by email, update the "movie" field, and save the updated document to the database. To remove a user, use a query to find the document with the matching email and remove it from the ***"users"*** collection.

IV. Global Control Flow

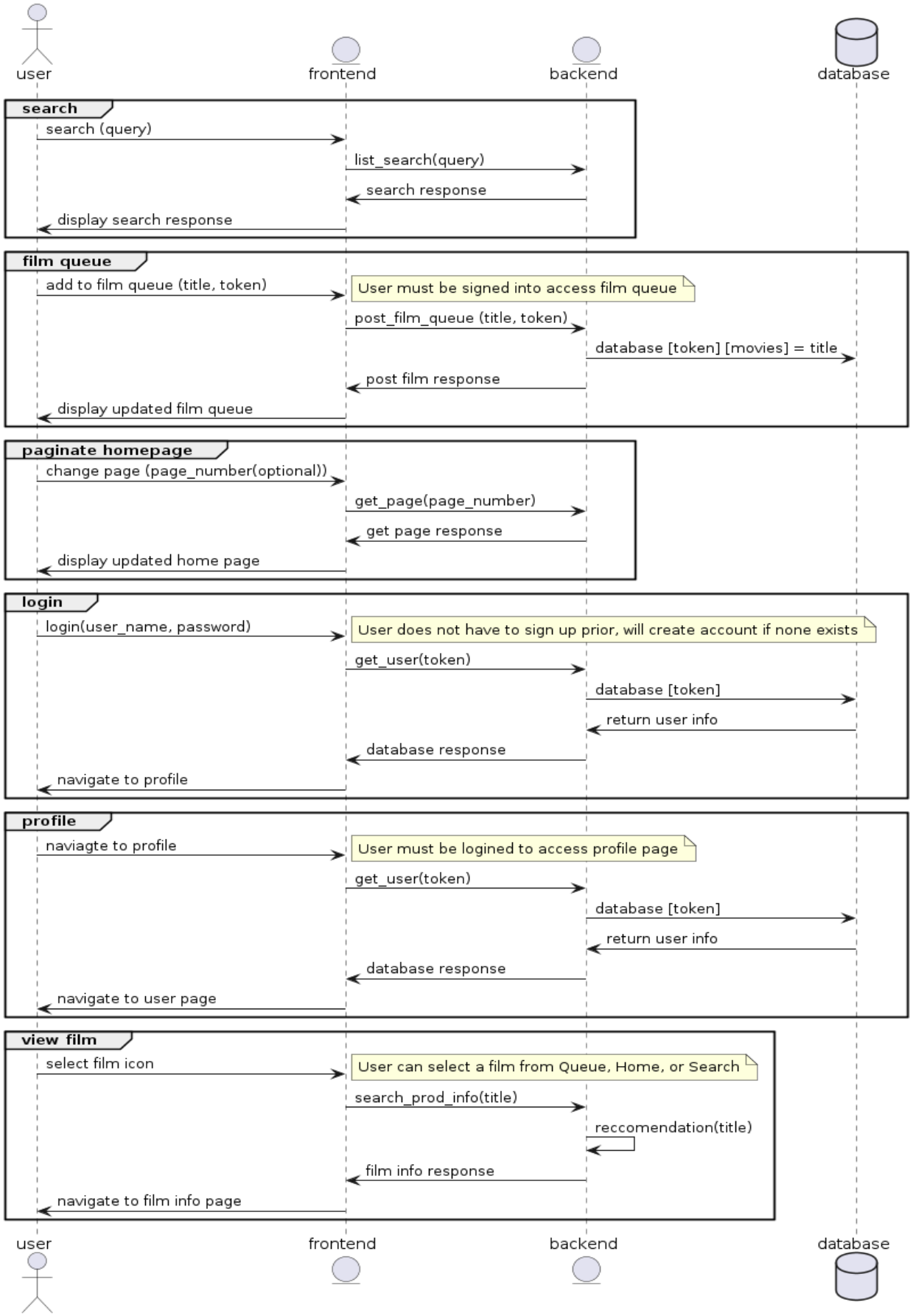
Our system is an event-driven architecture. The user has control over which elements of the site to access in which order. There are no time dependent features of the site. Instead, the system responds to user events in real-time, providing users with immediate access to information and recommendations. All of these architectural decisions reflect the site's purpose as a repository of information. User's should be able to navigate to specific information that interests them as quickly as possible. If elements or sequences were restricted in any way, the user would experience them as an inconvenience. The only element of the site that must be engaged linearly is the login process. Of course, the login process is expected to be engaged linearly, so it is not annoying to the user for that to be the case.

V. Detailed System Design - Will

A. Static View



Here we can see a static overview of the website. Almost all of the logic present in the site is contained in the 'Base' class. The base class functions as the only required file for a Flask application, so all of the api endpoints are contained here. The 'User' class is a representation of the database schema, so it is contained within the 'Base' file. Most 'classes' in this diagram are actually the frontend pages/components. 'Home' through 'Profile' are all react pages that are called by the main 'App' class. Navbar is a component that every page calls.


user
frontend
backend
database
search
search (query)
list_search(query)
search response
display search response
film queue
add to film queue (title, token)
User must be signed into access film queue
post_film_queue (title, token)
database [token] [movies] = title
post film response
display updated film queue
paginate homepage
change page (page_number(optional))
get_page(page_number)
get page response
display updated home page
login
login(user_name, password)
User does not have to sign up prior, will create account if none exists
get_user(token)
database [token]
return user info
database response
navigate to profile
profile
naviagte to profile
User must be logined to access profile page
get_user(token)
database [token]
return user info
database response
navigate to user page
view film
select film icon
User can select a film from Queue, Home, or Search
search_prod_info(title)
reccomendation(title)
film info response
navigate to film info page
user
frontend
backend
database

From the homepage, the user will have the options to access the homepage, search, and individual film information without having to login. All of these pages simply query the Flask server for film information, whether that be individual film info or paginated multi-film info. The user will also be able to login/signup, upon which their info will be synchronized with the Mongo Database via the Flask server. After they are synchronized (through the use of a token) they can now access their profile page, as well as their film queue. The film queue is persistently accessible from any page, and updating it requires once again updating the Mongo Database via the Flask Server.