CIS450/550 Final Project
Landon Chow
Zifan Dong
Ginni Fang
Le Liang
Tianyi Zhang

Netflix & Chill

A Movie Recommendation System

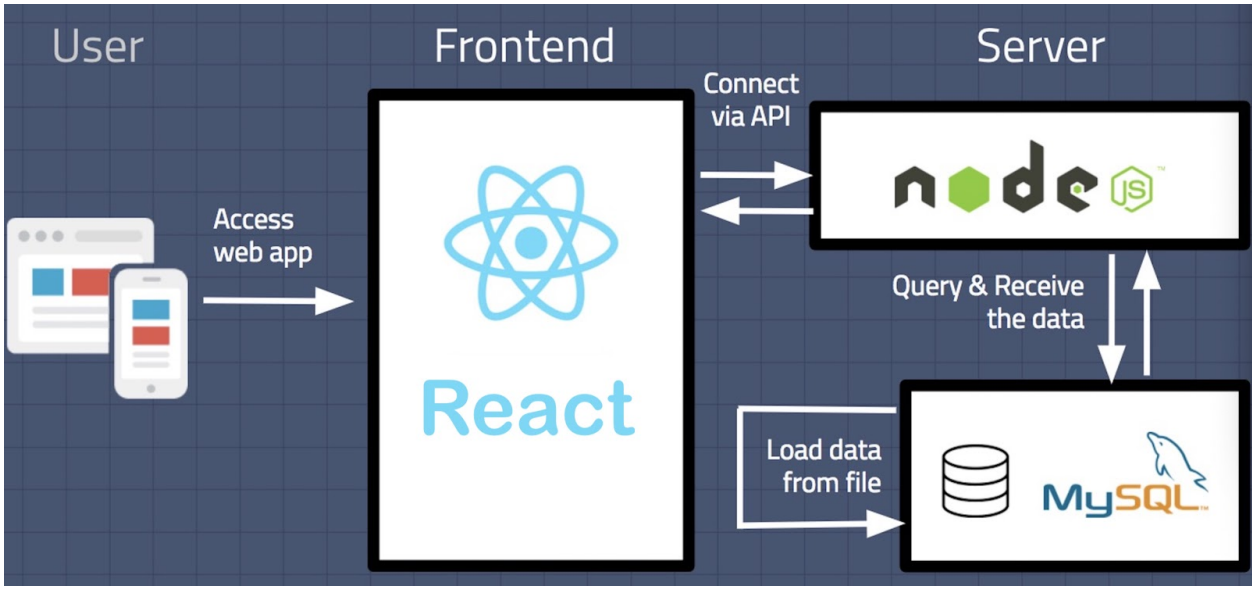
Introduction

You've spent so long fawning over that cute boy who always sits in the front of class and answers Professor Davidson's questions. You stomach the courage to ask him out, and he's agreed to come over for Netflix and Pizza this Friday! You spent the last few days cleaning out your apartment and even picked out an adorable outfit... But as Friday nears, one problem still remains...What movie should we watch!?

Snuggle is A practical, purpose-centric movie recommender - whether you're searching for the best movies (excuses) to curl up next to the guy of your dreams or the best movies to bawl out on by yourself after an emotional day, *Snuggle* is the go-to web app that will give you curated movie selections to suit your every need and mood.

Unlike traditional movie databases that only allow searches based on traditional filters like title, genre, and ratings, *Snuggle* provides *mood-* and *need*-driven movie recommendations by intelligently wrapping traditional database queries on the back-end inside creative "tags" on the front-end that better connect with users' actual needs. In addition to our main tag-based recommendations, we also provide a more traditional query tool to allow users to quickly search for movies based on more traditional filters - in case they are looking for something very specific.

Architecture and Technologies Used



The architecture of the system was designed with extensibility and modularity in mind. It was meant such that if one aspect of the system changed (database provider, database type), the amount of work necessary to bring the system back to full user functionality was minimized. To this end, the system was designed in two main parts--the frontend/server, and the content database.

The frontend was developed using React.js, a modern module-based framework originally developed by Facebook. The server was developed using Express.js, which acts as the intermediary between client and database services. It interprets the client requests and marshalled and modified them in the correct way such that they could be interpreted correctly by external database services. The React.js and Express.js services are hosted on a Windows Server 2016-based AWS EC2 instance. To manage the different database interactions, Express uses the Mysql service to make requests to our AWS endpoints. When a search query is performed, the server calls a file which performs the search and prepares a JSON with the results to pass to the client.

The second component of the system is the content database. It is here that our cleaned, indexed, and separated data about movies is stored and accessed. The database is of MySQL 5.7 type running on an AWS managed RDS instance. Our original datasets were in csv and txt formats which we cleaned using a combination of python scripts, Microsoft Azure's data cleaning service, and Excel, and then imported into the AWS RDS instance using the Table Data Import Wizard feature of MySQL Workbench. The queries used to access these datasets are all SQL queries.

Data Sources and Relations

Although it is relatively easy to find datasets about movies, it is hard to find ones that fulfill our need the most. After carefully searching and several attempts of cleaning and filtering, we have used the following two datasets:

- The Open Movie Database(OMDb API)(link) - Can be joined with Kaggle dataset via IMDB ID.
- Kaggle- The Movies Dataset (link) - Contains a large amount of metadata on 45,000 movies and 26 million ratings. Contains movie plot descriptions not included in the above dataset.

After we analyzed and cleaned the data source and tried multiple data types including .csv and .txt, our normalized relations schema is as below:

Actors (IMDB_ID, Actor)
IMDB (IMDB_ID, director_name, duration, gross, movie_imdb_link, country, content_rating, title_year, imdb_score)
Kaggle (IMDB_ID, id, overview, poster_path, title)
Genre_Processed (IMDB_ID, Genres)

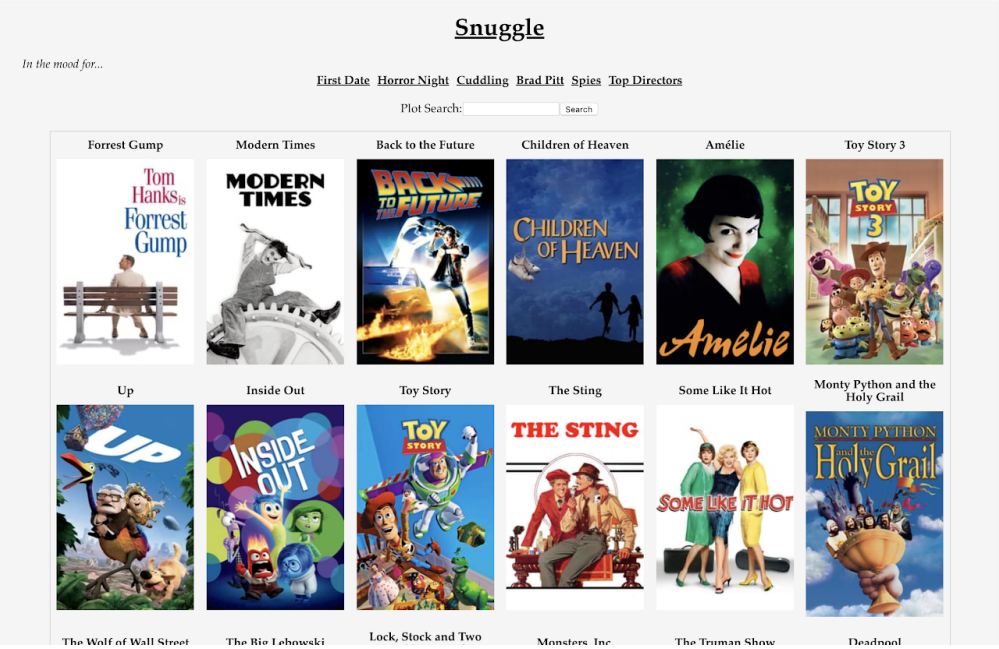
Main Features

Frontpage

Currently, the demo version of *Snuggle* has a front page shows all its functionalities, including getting movie recommendations by mood, traditional movie search, and Plot Search. Below is the frontpage of *Snuggle*:

Get Movie Recommendation by Mood

Currently, the demo version of *Snuggle* offers preset tags of different moods, including First Date, Horror Night, Cuddling, Brad Pitt, Spies, Top Directors. In the future, more user-oriented tags will be implemented and added. Below is the page showing all movies we recommended for first date after the user click First Date:

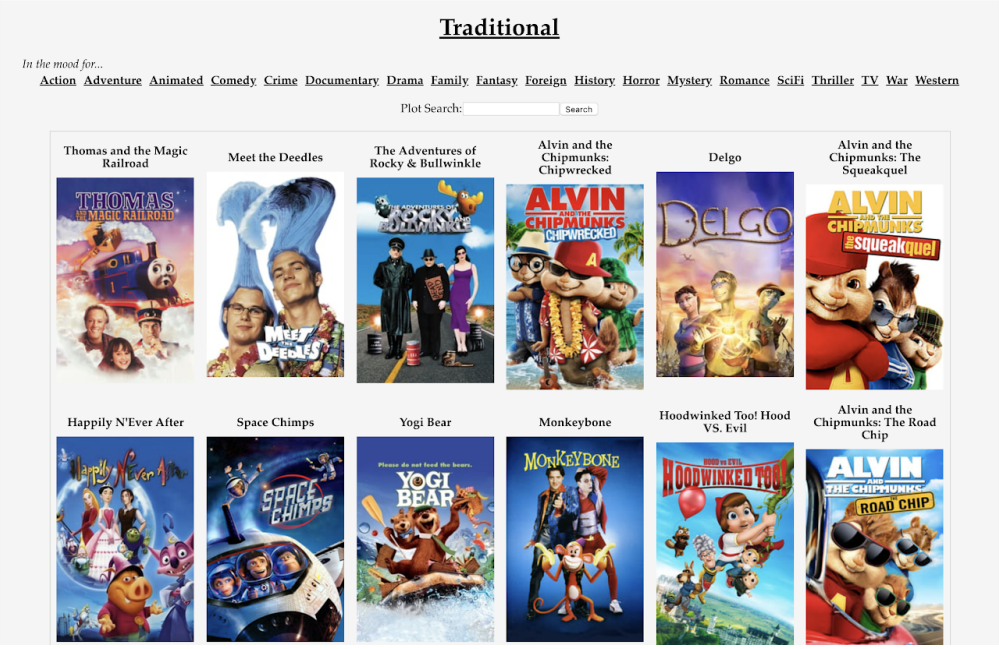


Different collections of movies will show on page when different tags are clicked.

Traditional Movie Search

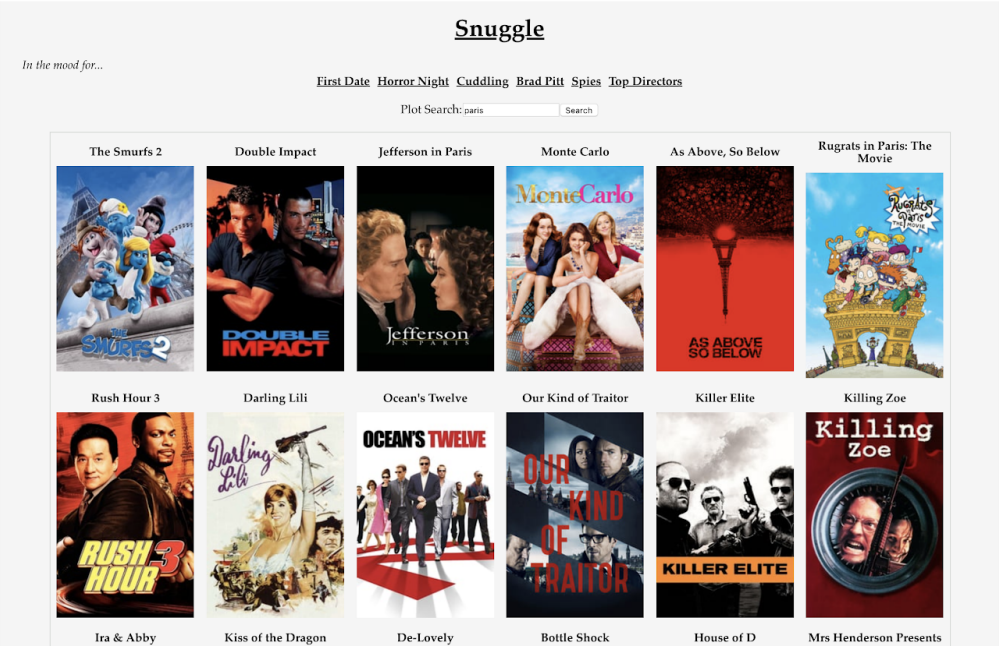
After users click the Traditional Search on the lower right corner of the page, they will see a page of traditional search which allows users to get movie recommendation when they know what kind of movie they want to watch. Movies are categorized into 12 genres, all of which is shown on the page for users to click. Below is the main page of traditional movie search:

And below is a page after Animated is clicked:



Plot Search Engine

Below the navigation bar, we implemented a search box which allows users to search for movies by plot. We feel that this feature is extremely useful because oftentimes what users have in mind may not be captured by traditional filters like genre, actor, etc. For example, a French teacher might want to show his/her students a movie about Paris, but no movie database has something as specific as Paris as a genre. Using *Snuggle*, he/she will be able to simply search for "paris" in our plot search bar, and see a list of highly-rated movies about Paris:



Technical Considerations

Data Cleaning

The two datasets we finally decided to use were raw and needed massive cleaning efforts. Specifically, Kaggle database contains metadata on 45,000 movies and 26 million ratings. It also contains the plot descriptions which is useful information for us to build our website. For the OMDB database, it has more robust information on popularity, runtime, actors, etc.

We joined these two datasets by using the imdb_id, which is consistent and unique in both datasets.
Then we used Azure, an online cloud platform to pre-preprocess the raw data, including cleaning missing data by interpolation, replacing abnormal data with mean value, visualizing the data for quickly extracting and analyzing, ect. We also wrote some code for handling the issues faced in the preprocessing stage that were not handled by Azure.

In order to address required features and improve the searching performance, we divided the two datasets into four relations (as shown above), and further cleaned and processed the data so that they are accurate and useful. We wrote unique Python and MatLab scripts tailored to the dataset to read in all rows, store values in a dictionary or a set, and then check for missing or poorly formatted values. Below is a summary of specific data formatting issues we encountered and how they were solved:

Issue	**Solution**
Duplicate entries	Eliminated duplicate entries
Data importing error: comma is both the divider and part of the content in a .csv file	Generated SQL INSERT method for with special characters and replaced characters after
Special characters, empty or null values	Cleaned all such invalid data using Python, SQL queries, etc
Abnormal data value	Replaced with mean value using Azure

Performance-based database design

We also designed our database so that query-writing and processing can be simpler for future updates. Specifically, we separated the original datasets into four main tables, carefully choosing the attributes we need based on both current and projected future features, and reorganized attributes into new tables in order to improve performance (as shown above in Data Sources and Relations). For example, genres were originally structured as a list attribute as part of a huge dataset (e.g. { IMDB_ID=123, Genre=[Comedy, Romance,

Drama], ...} but considering we have a traditional movie search function which needs to query genres a lot, we separately created a table for genres (setting genre as part of the primary key index), and connected it with other tables using IMDB ID. Then new table stores {genre (single), IMDB} pairs which enables significantly faster queries on genre.

As another example, one of our queries was to select all the movies featuring both Christian Bale and Tom Hardy. In the original Actors table, one IMDB ID identified three different actors (in separate tables). If we use the original data structure, we would need to query every pair-combination of (Actor 1, Actor 2, Actor 3) to give us the result we wanted. We decided to preprocess the Actors table by indexing it so that it has only two columns, where each IMDB ID is with just one actor.

Complex SQL Queries

Below is a sample complex query used to select movies featuring both Christian Bale and Tom Hardy. We used multiple joins and subqueries.

```
SELECT IMDB_ID, movie_imdb_link, title, poster_path, overview
FROM IMDB NATURAL JOIN Kaggle NATURAL JOIN Actors
WHERE Actor = 'Christian Bale' AND IMDB_ID IN
(SELECT IMDB_ID
FROM Actors
WHERE Actor = 'Tom Hardy')
ORDER BY imdb_score DESC;
```

Another example in which we've used aggregation is our "movies by top directors" tag. Originally, we used a temporary table to store the top directors, and then queried movies by directors in the temp table.

```
CREATE TEMPORARY TABLE DIR (SELECT director_name, SUM(gross)
FROM IMDB GROUP BY director_name
ORDER BY SUM(gross) DESC LIMIT 3;

SELECT DISTINCT B.IMDB_ID, A.movie_imdb_link, B.title, B.poster_path, B.overview
FROM IMDB A JOIN Kaggle B ON A.IMDB_ID = B.IMDB_ID JOIN DIR
WHERE A.Director_name = DIR.Director_name
ORDER BY A.IMDB_Score DESC;

DROP TABLE DIR;
```

However, we realized that creating and dropping the temporary table is extremely slow, so we ended up caching the result of this query in memory (to be discussed in the next session). Had we not using caching (which allowed for the highest performance improvement), we could have also improved performance by converting the temporary table into a subquery as follows:

```
SELECT DISTINCT B.IMDB_ID, A.movie_imdb_link, B.title, B.poster_path, B.overview
FROM IMDB A JOIN Kaggle B ON A.IMDB_ID = B.IMDB_ID
WHERE A.Director_name IN
(SELECT director_name, SUM(gross)
FROM IMDB GROUP BY director_name
ORDER BY SUM(gross) DESC LIMIT 3)
ORDER BY A.IMDB_Score DESC;
```

Performance Considerations

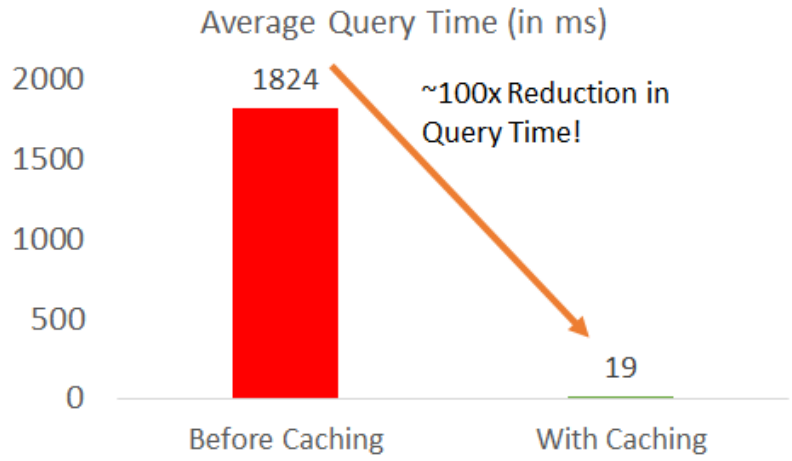
Given our database includes over 4000 movies (after data cleaning), performance, specifically query speed, is an important part of the overall user experience. Even after we finished our website and got the main features working, we weren't satisfied with the average query speed of 1824ms, which was far too slow to give the user a good experience. Although we had designed our database with performance in mind, even simple queries such as genre-based search still took over a second to complete, and more complex queries such as plot searches and top directors sometimes took well over 10 seconds to complete.

Full-Text Indexing

We drastically improved our slowest queries (plot searches) by using a form of special indexing not taught in class. We learned online that MySQL supports a special reverse-tree based index called full-text indexing which supports highly-efficient natural language-based searches. We previously had been manually scanning through each plot description (very long string) word-by-word for the searched keywords, which often took 10+ seconds to complete. After implementing full-text indexing, we reduced our average query time to roughly 2 seconds.

Caching

Given the majority of our search features involve fixed queries, we thought that caching our query results to LRU could drastically improve performance. The first time a *Snuggle* tag is clicked after the server starts, the server has to query the MySQL database to get the result, which results in a latency of ~2 seconds on average. However, after the first click, query results are saved to the LRU, and fetching that data takes just ~18ms on average. This improved performance by a factor of 100, which allowed us to give users the kind of seamless experience we envisioned.



Technical Challenges

We experienced many technical challenges which we had to work together to find ways to overcome. Some of the more notable challenges we experienced are discussed below.

The first problem we faced involved special characters when uploading data through MySQL WorkBench to our MySQL database. For some reason, MySQL WorkBench does not accept text with special characters (such as accents, or commas, when the divider is also a comma), and as such when we were importing a csv file, either the data import wizard repeatedly crashed, or the data imported was not accurate. We tried many different ways, such as using LOAD DATA INFILE statements, replacing the special character with other characters, and using IMPORT statement, etc. After several attempts, we were able to successfully imported accurate data into our database using a combination of special character replacement and manual formatting.

Second, we used React.js for the development of the website because we wanted to learn one of the most popular front-end frameworks in modern web development. However, none of us is familiar with React.js, or Javascript for that matter. We started from scratch. There was a moment when we got stuck for hours not being able to figure out how to make the GET and POST requests work between the server and client. Eventually we figured it out and realized there is another FETCH function that's much easier to use in our case. The process was painful, but we found it ultimately also very rewarding.

Future Features

As *Snuggle* is currently a simple demo of our idea, the functionalities are certainly expandable and has incredible potential. For example, we can eventually enable account creation to collect the user's demographic data and ratings toward movies he/she watched. Then we can implement a recommendation system using Collaborative Filtering or other algorithm to make personalized recommendations. We can also improve the traditional search

feature by creating more sub genres so that people can get movie recommendation more accurately. Finally, our front-end interface is fairly basic right now, and with more time we can certainly improve it to be more modern and interactive.

Extra Credit Features

- We hosted our database on AWS, and our server on Amazon's EC2 instance.
- We have image fetching of preset sizes for movie posters from the IMDB website.
- We used Natural Language Full-Text Searches of MySQL.
- Sophisticated technologies used: MERN Stack (MySQL + Express + React + Node.js).