# Movie Recommender

ramkumr2, hskhan10, xzhou45, amthoma3, hoyinau2, ziweiba2, zwen6, wmaanxg
04/30/17



# *Overview*

Movie Recommender is a service that gives users movie recommendation based on numerous different parameters such as genre, rating and cast. The user interface is based around a chat agent, which is built on top of a service provided by api.ai. Our service also allows a user to keep a profile of their watched movies and a separate learning agent provides customized recommendations based on a user's watch history. Finally, our service provides movie showtimes information and streaming options, if applicable.

# *Process*

Our team followed the Extreme Programming (XP) process. XP is a development process that develops and modifies the software according to the responses from customers frequently. See https://en.wikipedia.org/wiki/Extreme_programming for more information. Before implementation, our team came up with a set of use cases and user stories. We followed XP by iteratively implementing each feature and altering code based on changing requirements. We met with our TA each iteration to adjust user stories according to his response. Over time, new use cases came about but we made sure to refrain from developing these features until they were needed. Our team also heavily used pair programming on both the front-end and back-end. Front-end was responsible for user interface implementation, gathering identifying information from the user and sending it to the back-end. Back-end was responsible for analyzing received information and generating movie recommendations for the user. In between each iteration meeting, we had team meetings on every Wednesday for creating our iteration plan and checking whether the requirements of the current iteration are being met. We met with

our TA on Monday's every two weeks to present our project, and get some feedback. In terms of the development process, we tried to build a working product as soon as possible to minimize the risk. Therefore, we only conducted simple unit tests and manual tests before our working product came out. After that, we continued to write more tests and refactor the code base for handling edge case and improving performance. All team communication was done through Slack.

# *Requirements & Specifications*

## User Stories

1. A user wants to find a movie based on certain filters such as genre, cast and rating.
2. A user wants to find a movie similar to another movie.
3. A user wants a random movie recommendation.
4. A user wants to create an account to keep track of watched movies.
5. A user wants to add a rating to a movie they have watched.
6. A user wants to view movies they have watched before.
7. A user wants personalized movie recommendations based on their watched movies.
8. A user wants to know the showtimes and reviews for a movie.
9. A user wants to know the streaming options for a movie not in theaters.

## Use Cases

### Use-Case #1

The user is able to get a movie recommendation based on certain filters such as genre, rating and cast. For example, if a user says "I want to watch an action movie with Tom Cruise", the chat agent will parse the relevant information, send it to the back-end, and the back-end will return the results. The user is also able to get a random movie recommendation and the current most popular movies are returned.

Primary Actor: User
Goal in context: Get movies playing in nearby theatres
Scope: User-visible
Level: (User) High-level
Stakeholders and interests:

- User: Get movie recommendations.
- Back-end system: Queries movie database.
- Front-end framework: Presents the list on the web front-page.

Preconditions:

- User initiated conversation with chat agent

Minimal guarantees:

- The user is notified in case there are no movies that fit filters

Triggers:

- User initiated conversation with chat agent

Main Flow/Main Success Scenario:

- A list of movies that fit the filters given [E1]

Subflows:

- None

Alternate Flows

- [E1] Chat agent could not respond with a movie suggestion

## Use-Case #2

The user is able to get a link to the fandango page for a movie if the movie in in theaters. The fandango link allows a user to find showtimes for a movie and buy tickets if they want.

Primary Actor: User
Goal in context: Get fandango link for movie
Scope: User-visible
Level: (User) High-level

Stakeholders and interests:

- User: Get fandango link
- Back-end system: Queries Guidebox API to get link.
- Front-end framework: Presents the link next to the movie details.

Preconditions:

- User initiated conversation with chat agent

Minimal guarantees:

- The user is returned a fandango link

Triggers:

- User initiated conversation with chat agent

Main Flow/Main Success Scenario:

- A movie, or many, that the user might like to see

Subflows:

- None

Alternate Flows:

- None

## Use-Case #3

The user is able to find a movie similar to one they have watched before or heard about. The user simply provides the name of the benchmark movie and the back-end returns the similar movie.

Primary Actor: User
Goal in context: Get similar movie
Scope: User-visible
Level: (User) High-level

Stakeholders and interests:

- User: Get a similar movie
- Back-end system: Queries database
- Front-end framework: Presents the similar movie on the web page

Preconditions:

- User initiated conversation with chat agent

Minimal guarantees:

- The user is notified in case there are no movies found

Triggers:

- User initiated conversation with chat agent

Main Flow/Main Success Scenario:

- A movie similar to the one the user provided

Subflows:

- None

Alternate Flows

- Chat agent could not respond with a movie suggestion

## Use-Case #4

For a suggested movie, the user is able to view all options of how one can either stream the movie online (or equivalently view options to purchase it online). The application displays a list of streaming options once a suggested movie has been agreed on and selected by the user.

Primary Actor:  User
Goal in context: View streaming options for suggested movie
Scope: User
Level: User (high-level)
Stakeholders and interests:

- User: Get a list of services through which a user can stream or purchase a movie
- Back-end system: Queries data based on movie title and returns a list of streaming/purchase options in JSON format.
- Front-end framework: Presents the list on the web front-page

Preconditions:

- Chat agent must have suggested a movie to watch
- User initiated conversation with chat agent

Minimal guarantees:

- The user is notified in case no streaming options can be found for the selected movie

Triggers:

- User initiated conversation with chat agent resulting in a movie suggestion

Main Flow/Main Success Scenario:

- User initiates conversation with chat agent
- The conversation results in the chat agent suggesting a movie
- Streaming options for the movie are displayed on the same web-page [E1 E2 E3]

Subflows:

- None

Alternate Flows

- [E1] Movie has no streaming options
- [E2] Chat agent could not respond with a movie suggestion
- [E3] Chat agent responds with an invalid/incorrect movie suggestion

## Use-Case #5

For a user input regarding preferences, such as the list of movies a user likes/dislikes, or the set of genres the user likes/dislikes, the learning agent will be able to use that information to update the learning model that is specific for the user.

Primary Actor:  Learning Agent
Goal in context: Modify learning model based on user input
Scope: Backend System
Level: sub-function
Stakeholders and interests:

- Learning Agent: Get the list of modifications from chat agent. Then modify databases and rebuild the model.
- Chat Agent: give input to the learning agent, then analysis the reply from the learning agent.

Preconditions:

- The chat agent handles the user request as input and give the correct command to the learning Agent

Minimal guarantees:

- The learning Agent will reply to the chat agent the result of the operation.

Triggers:

- chat agent calls the Learning Agent

Main Flow/Main Success Scenario:

- User initiates conversation with chat agent
- Chat agent gives the correct command to the learning agent
- Learning agent replies back

Subflows:

- None

Alternate Flows

- [E1] Chat agent gives a wrong command to the learning agent
- [E2] Learning agent accounts error during operations
- [E3] Chat agent fails to handle the reply from learning agent

## Use-Case #6

The learning agent will be able to sort the candidate list which is provided by the chat agent according to the correlation of user preferences from high to low. Then the learning agent will reply to the chat agent with the top five highest correlated movies.

Primary Actor:  Learning Agent
Goal in context: Compute the user model and sort the given candidate list, then reply with the top five on the sorted list.
Scope: Backend System
Level: sub-function
Stakeholders and interests:

- Learning Agent: Get the list of candidate moves from the chat agent, sort the list and reply to the chat agent.
- Chat agent: Prepare candidate list and make the right call to the learning agent, then analyze replied result.

Preconditions:

- The chat agent prepares the correct candidate list and user identifier and give the correct command to the learning Agent

Minimal guarantees:

- The learning Agent will reply to the chat agent the result of the operation.

Triggers:

- Chat agent calls the Learning Agent

Main Flow/Main Success Scenario:

- Chat agent prepares a candidate list.
- Chat agent gives the correct command to the learning agent with the list
- Learning agent computes the mode
- Learning agent sorts the list.
- Learning agent reply to the chat agent.

Subflows:

- None

Alternate Flows

- [E1] Chat agent gives a movie list that contains error
- [E2] Learning agent accounts error during operations
- [E3] Chat agent fails to handle the reply from learning agent

## Use-Case #7

The user is able to give the system a certain movie name and add the movie to his / her watched movie list, which will then be stored in the database.

Primary Actor: User
Goal in context: Add a movie to the user's watchlist
Scope: User-visible
Level: (User) High-level
Stakeholders and interests:

- User: Add movie to list of watched movies
- Back-end system: Add the found movie entry to the user's database table
- Front-end framework: Give feedback based on whether the operation has succeeded, and re-display the title of the added movie to the user.

Preconditions:

- User initiated conversation with chat agent

Minimal guarantees:

- The user is notified in case the movie the user chooses does not exist
- A movie will not be added to the watched list twice

Triggers:

- User initiated conversation with chat agent
- User adds the movie to the watch list

Main Flow/Main Success Scenario:

- A movie fit with the user given name is found [E1]
- The movie entry found is added to the user's list [E2]

Subflows:

- None

Alternate Flows

- The movie can not be found [E1]
- The movie entry is already in the user's list [E2]

## Use-Case #8

The user is able to view his or her list of watched movies.

Primary Actor: User
Goal in context: The user is able to view the watched list and quickly rewatch his or her favorites.
Scope: User-visible
Level: (User) High-level
Stakeholders and interests:

- User: View watched list
- Back-end system: Query the user's personal watched list
- Front-end framework: Display the watched list received from the server to the user

Preconditions:

- User initiated conversation with chat agent
- User has at least one movie in the watched list already

Minimal guarantees:

- The user is notified in case there are no movies in his / her list

Triggers:

- User initiated conversation with chat agent
- User asks for watched movie list

Main Flow/Main Success Scenario:

- The user's watched list is queried and returned [E1]

Subflows:

- None

Alternate Flows

- The watched list is empty [E1]

# *Architecture & Design*

## Front-End/Back-End Architecture

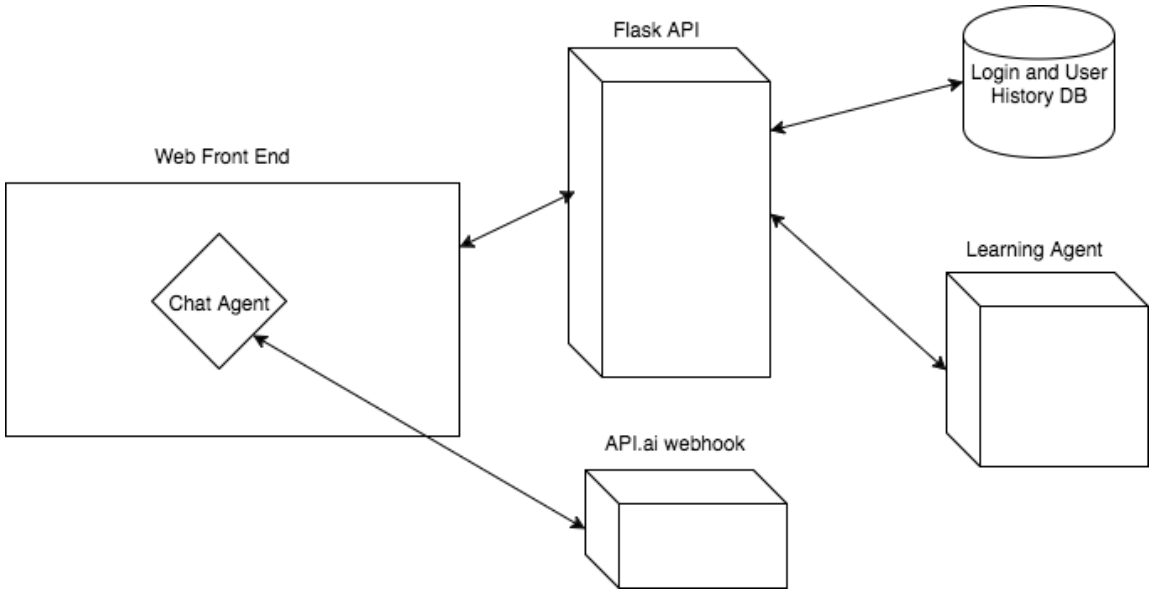


*Fig 1. See "Learning Agent Architecture" for more information regarding Learning Agent*

### Frameworks & Design Choices

For the front-end, Angular.js was the framework of choice. We went in this direction because it is a popular choice for dynamic web applications with minimal ramp-up time to deploy in our Heroku cluster. Angular.js allows easy integration with the front-end HTML templates through partials and directives which was another reason for our choice. Lastly, the framework has a lot of documentation and support available online which was a hugely beneficial for us as we had limited experience with web development. For front-end testing, we used Jasmine, a testing framework for Angular.js. We were able to test our controllers with easy-to-write unit tests, and mock any part of our application for easy testing purposes. For example, we could simulate an HTTP Post call to the backend and see if we get the right response, or use mocked objects to check if our API would process those correctly. For the back-end, we chose to use Flask. In short, Flask is a microservice for Python applications. We chose Flask because it is simple to build an API, as well as a front-end. Flask also integrates well with Angular.js and Bootstrap. More information on Flask can be found at http://flask.pocoo.org/. To test our Flask classes, we simply used the standard testing framework built in with Python. We constructed sample json request data and used that to simulate parameters to our functions.

## Front-End Code Structure & UML Diagrams

- static/js/app.js - Contains all Angular.JS states and controllers which control each aspect of the web application. Details in Fig 1.a.
- static/js/services.js - Contains the primary user service which handles all HTTP request with the backend flask API (webapi.py). The Flask API in turn helps communicate with the chat agent, learning server or database.
- static/partials/*.html - Contains all HTML files which are used by each Angular.JS state. For example, we have separate html files for the sign-up page, movie-detail partial and view-profile page.
- webapi_utils.py - Contains utility functions to communicate with the underlying postgresql database.
- model.py - Contains database model information for user database.
- static/js/app.spec.html - Contains the skeleton for the jasmine.js front-end tests
- static/js/app.spec.js - Contains the test cases for Jasmine controller/routing tests
- static/js/karma.conf.js - Contains setup for test coverage file

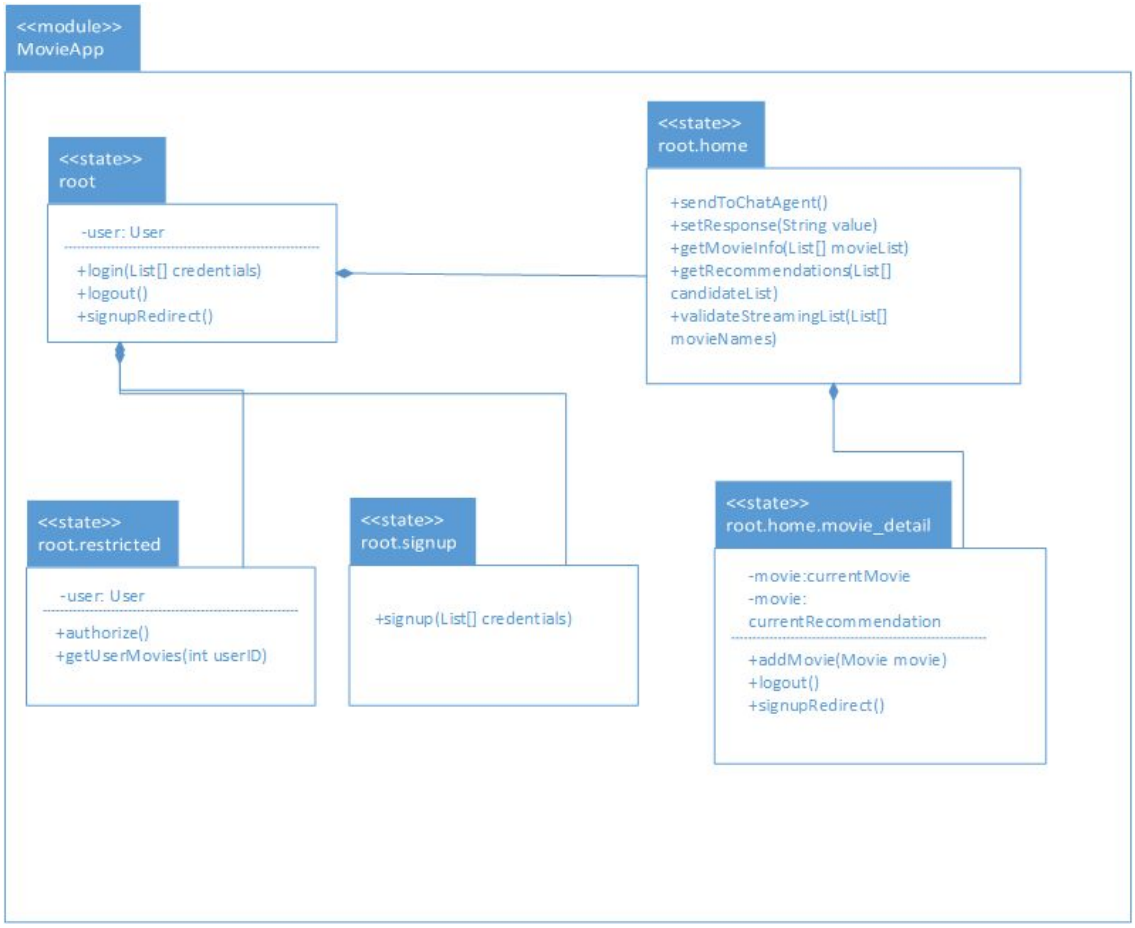


*Fig 2. Front-end UML Diagram*

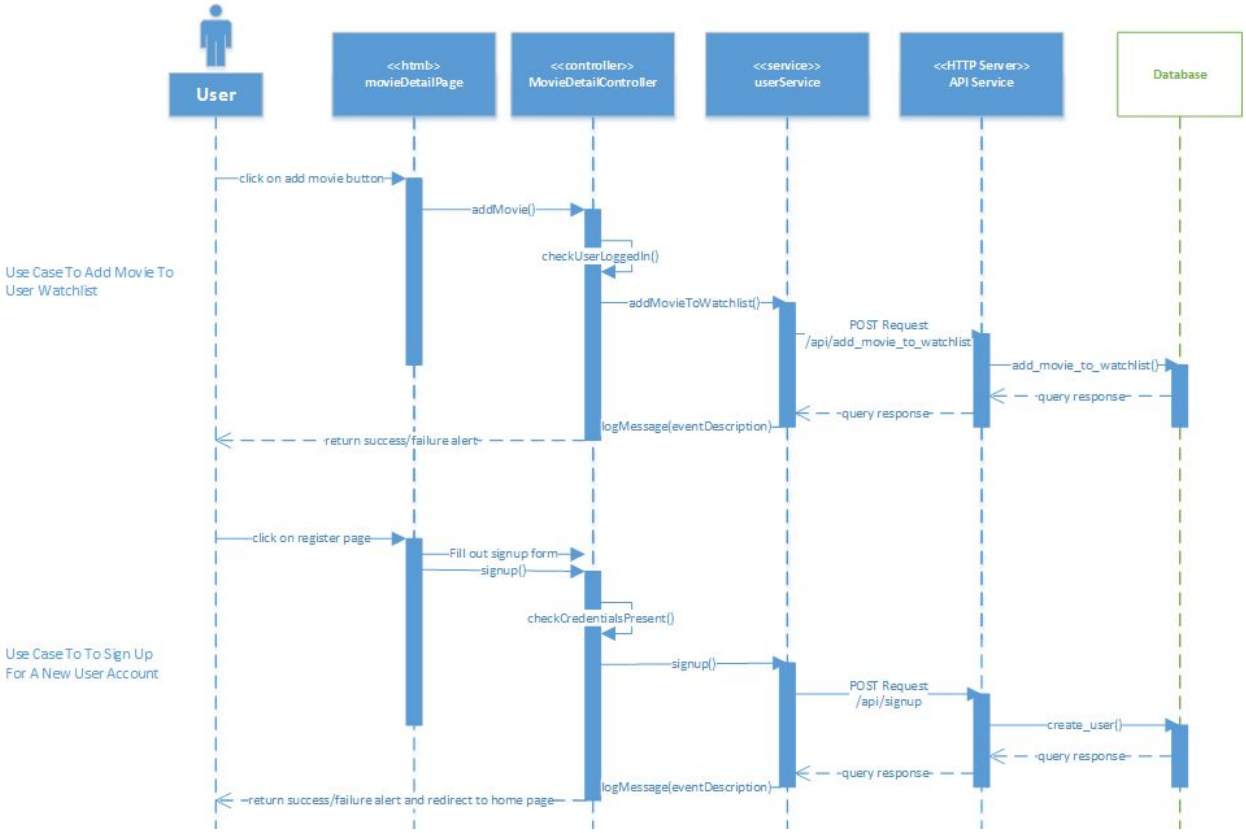


*Fig 3. UML Sequence Diagrams for Front-End User Story #4 And #5*

## Back-End Code Structure & UML Diagrams

- app.py - Contains 1 flask route which is hit by the api.ai chat agent.
- app_utils.py - Contains utility classes (MovieABApiClient)
- and functions used in app.py webapi.py - Contains API which is used by the front-end
- webapi_utils.py - Contains utility classes (LearningAgentClient) and function used in webapi.py
- model.py - Contains database model information for user database.
- tests.py - Contains unit tests for the chat agent functionality as well as back-end API calls.
- tests/ - Contains mock json request data which is used in tests.py

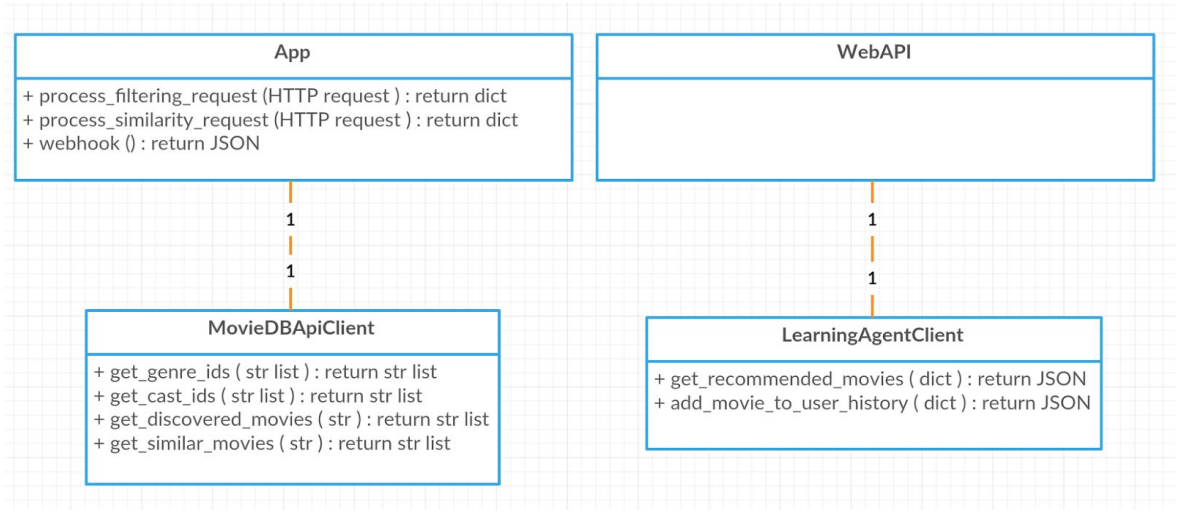


*Fig 4. UML Class Diagram for back-end code.*

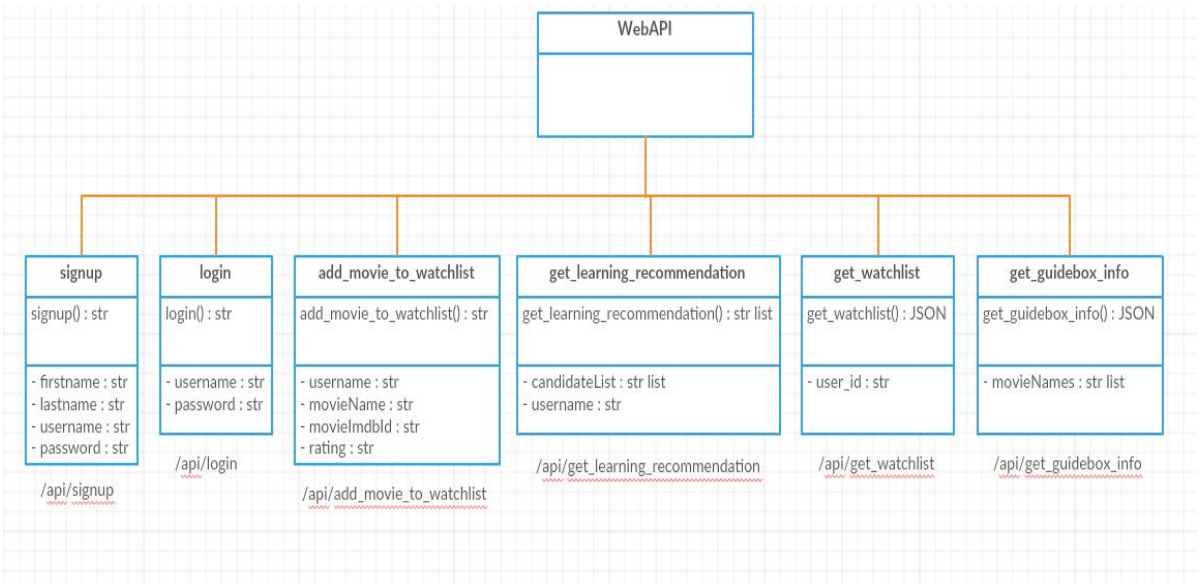


*Fig 5. UML Diagram for web api in webapi.py*

## Learning Agent Architecture

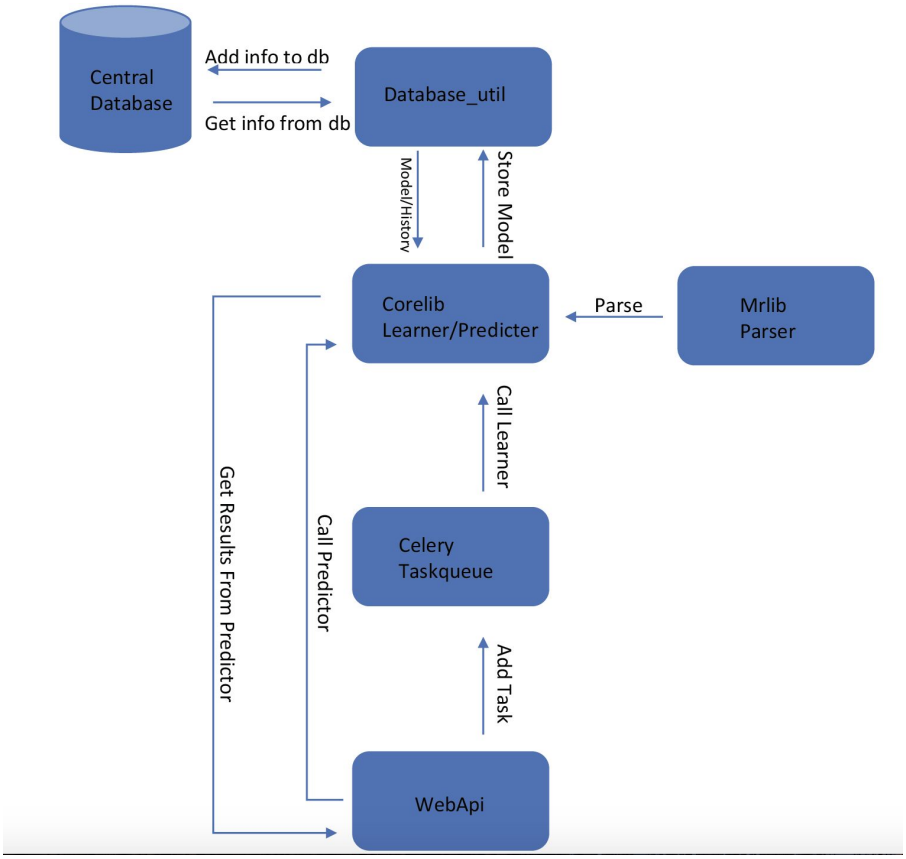


*Fig 5. Architecture for Learning Agent*

## Frameworks & Design Choices

We used a dataset from MovieLens, which contains about 27,000 movies. Each movies is associated with a 1128 long tag vector, which represents the "features" of a movie. We used this vector as the main feature vector of a movie, combining with Genres to achieve the final feature vector.

We used a SVM learner which predicts a binary tag (like / dislike) for every user on each movie. There is a score associated with the prediction so that we can rate and sort the candidate list that front end API passes with the likelihood that the user is going to like the movie.

We tested and tuned our learner using the real-world user ratings that MovieLens provides.

## Code Structure & UML Diagrams

- ./mrelearner/core

The core libraries for learning. Specifically it includes two libraries corelib.py and utillib.py.

corelib.py

The core learning library. It provides a class Learner and a class Predictor. Learner handles the training process for each user, and it is normally called when a new movie history is added for consideration for a user. Specifically, Learner reads the user history and train a new SVM model and then saves the model to database. This process is time consuming. Predictor handles predicts the score for newly input movies. It reads and loads the learning model directly from database and calculates the score. This process is relatively fast.

utillib.py

This is the utilities library to assist core learning. Mainly only one class Converter is used. This class handles the cases of transforming API caller ids to our internal ids.

- ./mrelearner/database

database_utils.py

This contains a database class which processes and handles all operations that are directly connected with the database. The current class relies on Microsoft ODBC driver.

parser.py

This file includes all the functions for establishing initial movie database tables.

- ./mrelearner/taskqueue

tasks.py

This file processes training jobs in a workqueue built on Celery. This ensures that the API agent is non-block so that all user requests can receive immediate return values.

- ./mrelearner/train

mrlib.py

This class was used mainly for tuning parameter purposes so most of the functions are not used in a working environment. However, some functions that helps building feature vectors are still in use.

- ./mrelearner/webapi

Web api for interfacing with the rest of the application.  Has one url for updating the user history and updating the learner model and one url for getting a recommendation from the predictor.

### Web API testing

The webapi was tested by manually issuing POST requests to the api. The following cases were covered: Updating user history with a brand new user, updating user history with a user with insufficient history to train a model, updating user history with sufficient history to train a model, getting recommendations from a user that is not in the database, getting recommendations from a user with insufficient history to train a model, and getting recommendations from a user with sufficient history to train a model.

### Core Learner/Predictor testing

The testing procedure is conducted via Travis-CI, the online unit tester and the coverage report is generated by Codecov. Due to the database driver we used (Microsoft ODBC) only works on Ubuntu 16+, and we did not find a working unit tester which properly supports this or newer Ubuntu versions. So, we created a pseudo `database_util` under `tests/`, which has exactly the same code as the mrelearner libraries, except that database_util now works on a local-file-base instead of requiring a driver. By this method, we are able to test everything except the database operations since there is no real database operations in the testing directory. The result is that we achieved a 96% coverage on core library and parsing library. Specific Report

### Database testing

As noted above, database testing can only be done locally. We wrote the testing script under the local directory which thoroughly tested every function of working database_util library. The test script also reach line coverage of 81%, which only corner cases are not covered. However, we still do manual test and checking for those corner cases (for example, when adding new movies into the database).

# *Reflection*

ramkumr2 - I learned how to properly follow a software development process. Following a process was helpful in planning and development of the project and it is something I can take with me into the workforce. Furthermore, I enjoyed starting a project from scratch and seeing it to completion. Finally, I felt like I gained good team management skills because I had to coordinate deliverables between multiple groups in our overall team.

hkhan13 - This was my first experience in trying to build a full-fledged web application, I learned a new language (javascript) and several frameworks (AngularJS + Flask) which was very valuable. Moreover, the concepts taught throughout the course helped with the way this project was organized and managed, especially regular meetings and maintaining meeting notes in a

wiki. The process we followed also emphasized regular and effective communication amongst the team.

amthoma3 - I learned about how to effectively self-organize and communicate goals and progress across a large team. I learned more about coordinating and processing API calls in a complex application. I definitely gained greater familiarity with git. Because our team was so large there were constant updates being pushed and a large number of branches. Staying up to date in your own work within your own branch necessitated frequent and carefully planned merges.

zwen6 - I learned the effective ways of reviewing other people's code, and how to write tests. Specifically I learned about Angular and Jasmine frameworks, and how the backend and the frontend integrate through data object interfaces. Also, in this medium-scale project the effective communication and frequent merging is extremely important.

ziweiba2 - I learned more about time management and working with unfamiliar frameworks which may or may not work on my local machine. I learned a lot about integrating my previous Angular knowledge with a backend system, and the setup of an app. I also learned about organizing my git branches so I didn't confuse my teammates later. Finally, I learned about the Jasmine testing framework and how to properly mock objects.

xzhou45 - This project helped learned how to properly plan for a project from the beginning to the end. I also learned how to manage and collaborate with team members, and schedule and assign works to everyone in the learning team. Another big part is I learned how to work with other team members who rely on my products: things like coordinating API calls and maintain services were new to me. Finally, testing is another big part that I enjoyed. We put a lot of effort into testing and using testing theories and do systematic testing for the first time has been educational.

mwang3 - On the technical side, I learned a new framework(flask) and gained experience working with cloud services like Azure.  I also learned new things about Git and the Python programming language.  On the non-technical side of things, I learned how to learn new things as I go and how to work as part of a team on a larger project.  Lastly I also learned the importance of testing and having thorough tests.

hoyinau2 - I learned a lot about how to apply machine learning knowledge into real applications, such as the concept of treating user preference as a vector and using support vector machine to predict probability to produce movie ranking. I also learned about using online service like Azure database, and the speed and response time of online computation. Apart from technical knowledge, I also know more about how people cooperate in a very fast pace on changing environment and fixing bugs of the code for working service.