Assignment 2 Design Report

Shaun Price - spri200
CompSci 235

New Feature: OMDB API

Use the Open Movie Database (OMDB) to access url for movie posters. Implemented using Python's `json` package to send requests, using the movie title and release years as query parameters to `http://www.omdbapi.com/?apikey=[apikey]t=[title]y=[year]`.

Received data is parsed into JSON format, attempt is to made to access the `poster` field. If a request has successfully been received, poster URL is returned for use in the CS235FLIX website. If request fails, then accessing the `poster` will throw a `KeyError`, from here the error is caught and image URL returned is a local `image-not-found.jpg` image.

Poster URL edited after being received to replace size info `300` in the URL to `600` to get the URL of a larger image. Returned poster URL is added as a key value pair in `movie_dict`. This, along with other movie data is passed through to the movie html page template and can be accessed using the Jinja templating language to dynamically display the movie poster image.

Design Decisions

Repository Pattern

Abstract repository implemented using Python's `Abstract Base Class (abc)` to define an interface that all repositories will inherit from. All accesses to the database repository from the service layer will be done through this high level module.

This pattern is especially relevant for this project as next, the in-memory repository will be replaced with a SQL database, The database repository for the SQL database will inherit from the abstract repository, because the service layer depends on the abstract repository, this makes it possible to implement a new database repository and have it be used by the service layer with no changes to the service layer itself. Making development faster and easier, a well as making the project more open to changes and extension.

Dependency Inversion

Repository pattern is an example of how dependency inversion principle was used in this project, instead of having a high level module (service layer) depend on a low level module (repository), we make it dependent on an abstraction, the abstract repository. It's easy to see that this makes it much easier to develop and extend the application when we will come to implementing a SQL database, since the service layer depends on the abstract repository, the service layer does not have to be aware of how the database is implemented, as long as any subclass of the abstract repository return the same type of data as what it expects.

Single Responsibility

Each module in the project has a well defined responsibility. Views are used to extract information from HTTP requests and build HTML pages. The service layer is responsible for acting a medium between the repository and view layer. It is role is to fetch data from the repository, process requests from the user and update the repository data. The repository is responsible for interacting with the database and dealing with data that is used by the service layer, it is not concerned with how this data is used.

This improves the design as the changes can be made within one layer with minimal or no affect to the others, making designing and building the application much easier and reduces entanglement between layers which is difficult to extend and debug if issues arise.

Separation of Concerns

Separation of concerns is seen in this project with the use of Flask blueprints. Blueprints allow the ability to organise application functionality into different, separate units. Blueprints should also follow the single responsibility principle and only be concerned with one part of the view. Different areas of the view layer are separated and can be worked on separately, making debugging and testing easier and faster, since you are working with smaller chucks of code and preventing the system from becoming tightly coupled. The home, genre, movie browsing, movie, and search page were all built completely separate of each other, maintainability and extendability by adding extra pages was easy as each page was decoupled from all the others.

Templating

Templating is an extremely useful tool to speed up development, ensure consistency and build reusable code. Templating was implemented in this project using the Jinja templating language. The most important application of this in this project was the development of a layout template, that defined the common look of the pages and functionality common to all pages such as the top and side nav bar. Templates could extend this parent layout template by changing the content block to each specific pages needs.

Testing

Unit and integration testing was used in this project. TDD allowed the creating of specifications to drive towards, creating a clear goal of what the expected behaviour of code is, and how it works within the entire system. Having tests allowed to quickly verify the code works after any changes and refactors were made in this project, such as refactoring the service layer to remove duplicate code and replacing them with new functions. This saves time and identifies bugs quickly. Testing was done using Pytest in this project.