Ho Yin Alvin Cheung

Due: 2020-10-25

UPI: ache706

COMSPCI 235 – Assignment 2 Design Report

A-Grade Creative Extension

I increased the required functionality by allowing users to edit and even delete their comments when logged in.

When a user logs in, the application creates two hyperlinks - one for editing, and the other for deleting - for any review written by the logged-in user. The hyperlink points to a URL based on the review's hash. When the user navigates to that URL, the app checks the session to see if the user has logged in, while checking whether the correct hash exists for the comment. Even in an unlikely scenario where hackers succeed to find out the timestamp, they cannot edit or delete the review because they don't have a valid session. The implementation satisfies the security requirement.

Principal Design Decisions

I applied the domain-based design model, test-oriented development, the principle of single responsibility, dependency inversion and the (abstract) repository pattern. We cannot use databases for this assignment.

Domain-based Design

I oriented my design to be a minimal-viable product. You may find out that there are little decoration and extra functionality other than the requirements. My approach ensures that I can produce a product as efficiently as possible, while still able to collect users' feedback. The domain model helps me in developing structured software without much deviation, provides unity and aids understanding for future developers.

I name the classes and methods base on the domain-based principle. For example, the Movie class represents a movie, which is a subclass of Media. I leave that in the final product for extra flexibility of extensions in the future because the Media class can represent any piece of digital information.

Methods among classes and relationships reflect aspects of the model that can be responsible in real life. To name but a few, the User class include methods "watch_movies" and "add_review", which represent frequent actions from a user.
The number of developers will grow when we continue to expand the software. By the responsibility-based naming principle of classes and methods, fellow developers can easily understand the application structure, rather than having to undergo the excruciating process of learning it from scratch.

Test-oriented Development

Test-oriented development is an iterative methodology where developers write and run tests throughout the development process, rather than after the developers complete developing the program.
I produced unit tests before completing the code in this context. Test-oriented development allows developers to anticipate possible logic errors, plan through implementation and consider several aspects of the project before developing the software. It leads to a design which has fewer logic errors and is sturdier.

Single-responsibility Principle

The single responsibility principle states that each module in an application ought only to have responsibility for one functionality. The service layer of my application is one of the examples.
The request handlers receive HTTP requests and respond with HTML pages. The service layer handles all application logic and decides what should include in the HTML pages - meaning that the service layer checks the session, requests required data from the repository, and gives a list of data for the request handlers to display.
Use of the single responsibility principle means that we can minimise dependence between classes. As I will continue in Dependency Inversion, when we need to change a class, single responsibility principle means that the changes will not affect the other modules' responsibilities. The reason behind that is the class or method has only one responsibility.

Dependency Inversion

Dependency inversion utilises abstract interfaces to disconnect the high-level modules from the low-level modules. I separated the implementation of the repository from the service layer to apply this principle.

We need to replace the memory repository with an SQL database in Assignment 3. Because we use dependency inversion, there is already an existing interface in terms of the Abstract Repository class. Thus, changing the database implementation will not alter any function calls in the service layer.

Repository Pattern

There is an abstraction layer on top of the data access layer. The repository pattern also consolidates the handling of data requests. My application uses an abstract interface as the front end to access the memory repository, as mentioned in the "Dependency Inversion" section. Abstracting the implementation with a repository pattern provides a single interface with straightforward methods.

The application now uses only one database. However, if we need more databases in the future, the repository pattern enables us to interact with all databases by the existing methods. The design simplifies the simulation of the database during testing. We can even replace the existing memory-based repository with an SQL database.

Final Notes on Security

Developers usually update their software to patch security issues. Thus, to maximise security, I updated all dependencies in requirements.txt to the latest version possible without breaking the code.