Project 7
CSCI 5448

CafèConnect

Jayant Duneja
Jayant.Duneja@colorado.edu

Tilak Singh
Tilak.Singh@colorado.edu

December 10, 2023

Final State of the System Statement:

Summary

We have created ***CafeConnect,*** an application where users can view their favorite restaurants and interact with them by leaving reviews. The users can also interact with each other and connect with other users through friend requests. Finally, the cafes can send notifications to all the subscribed users about special deals and discounts that are present at the restaurant.

For this, we have majorly used an MVC approach for this project; we have various models and have corresponding views and controllers for interacting with them.

Lifecycle of a Request:

- Whenever we get an API request from Postman, the request is received by the controller. This is where we manipulate the request to get the data we need.
- This data is then forwarded to the Service which has all of the business logic for our application. The service decides what data should be extracted from the SQL database. The service calls the repository which extracts runs the SQL queries on the database
- The repository runs the SQL query to extract the data from the database. The outputs of this query are verified with the model we have defined in our code to make sure that the schema of the model class and the tables are matching.

Models

- These are the following models present in our code which map to real world entities:
    - Cafe
    - Menu
    - Review
    - Student
- These are the composite models we created to implement some of the features:
    - CafeStudent
    - FriendNetwork

Features Implemented:

- Database Management: Our application relies on CockroachDB via Spring Boot, ensuring efficient and secure data storage.
- Cafe Exploration: Students can easily discover nearby cafes, enabling them to explore restaurant pages and peruse the available menu items.
- Menu Customization and Discounts: Cafes have the flexibility to enhance their menu by adding items and applying discounts, providing students with a dynamic and enticing dining experience.
- Review System: Students can share their dining experiences by leaving reviews, fostering a transparent platform for feedback. Restaurants can conveniently access and respond to customer reviews.
- Notification Subscriptions: Students have the option to subscribe to notifications from their favorite restaurants, receiving updates on special deals and offers. Restaurants can effortlessly engage with subscribed users.
- Interactivity and Social Features: Students can connect with peers by sending and receiving friend requests. This feature enhances the social aspect of the application, allowing users to build connections within the community.

Features not implemented:

- We have not implemented login for any user. Due to this, we were not able to implement the following features on the frontend(UI):
    - We were not able to give the option to the user to leave a review through the front-end application since we needed the particular unique user-id to record that.
    - Similarly, a student does have the option to send a friend request or subscribe to a restaurant from the UI since we do not have a particular user-id for our frontend student landing page.

Changes from Project-5 and Project-6:

- In Project-6 we had mentioned that we would be having a Cafe view, which would be a UI showing the various items the cafe offers, the list of menu items for that cafe and all the reviews which multiple students have left for that Cafe. We were not able to implement this frontend component due to shortage of time

Patterns Used:

1. Singleton Pattern:
    - Purpose: The Singleton Pattern was employed in the connection to the database through the SpringBoot application.
    - Usage: The Singleton Pattern ensures that a class has only one instance and provides a global point of access to that instance. In this context, it was used for Lazy Instantiation of the Datasource, meaning that the connection to the database is created only when it is first needed.
2. MVC Pattern (Model-View-Controller):
    - Purpose: MVC was implemented for interacting with the tables (Cafe, Menu, Review, Student) in the database.
    - Usage:
        - Model (M): Represents the data and business logic of the application. In this case, models were created for each table (Cafe, Menu, Review, Student), defining the structure of the data and any necessary business logic.
        - View (V): Represents the user interface or the presentation layer. We used React.js to show the UI and for our project 6, we have implemented the `/student` route to show the student view.
        - Controller (C): Manages user input and updates the model accordingly. Controllers were implemented for each table to handle the logic of processing requests and updating the data models.
3. Observer Pattern:
    - Purpose : Our system incorporates the Observer pattern to meticulously log all events occurring within the system. This allows us to maintain a comprehensive record of system activities for monitoring and analysis.
    - Usage : Within our project architecture, we leverage the inherent Java Observer and Observable classes for implementing the Observer pattern. The Subject, responsible for emitting events, and the Observer, responsible for logging these events, both inherit from their respective Java counterparts. To seamlessly integrate this pattern into our services, we utilize Dependency Injection. The Subject is passed to services through their constructors, enabling services to efficiently log pertinent messages as events unfold within the system. This modular and extensible approach enhances the maintainability and flexibility of our logging system.
4. Strategy Pattern :
    - Purpose: Our system uses the strategy pattern to provide a discount on an item. This allows us to maintain a record of items with and without discount.
    - Usage : We implemented this by having a base interface `DiscountStrategy` and then we created classes `ItemWithDiscount` and `ItemWithoutDiscount` that implemented the `DiscountStrategy`.

OO Fundamentals Used:

1. Inheritance : Inheritance was used to implement the Strategy pattern.
2. Polymorphism: Polymorphism was used while defining the routes in the Controller
3. Abstraction: MVC pattern along with the repository pattern helps us implement Abstraction.
4. Encapsulation : All the variables defined in the Models were all private and they had public getters and setters for controlling access.

Final Class Diagram and Comparison Statement

CafeConnect UML (Project 7)
CafeConnect UML (Project 5)

Key Differences between UML (Project 7 v/s Project 5)

- In Project 5, we initially planned to utilize the Observer pattern for sending notifications; however, we ended up employing the pattern to create a logger for the project.
    - For notifications, we did not implement the Observer pattern because of the dynamic nature of the data we were working with.
- In Project 5, our initial intention was to apply the Factory pattern for adding menu items. Due to a lack of classification, we subsequently transitioned to the Strategy pattern to implement a discount class for the existing menu items.
- Besides the previous design pattern changes in UML, no further alterations were made to the planned pattern implementations. Nevertheless, we refined our approach to the implementation of User and Cafe. In contrast to Project 5, where both were extended from a common User class, in Project 7, we designed them as separate classes.

Third-Party code vs. Original code Statement

In executing the project, we refrained from utilizing external resources. However, for a deeper understanding and implementation of various design patterns and object-oriented concepts, we relied on class notes. Additionally, external sources like Refactoring.Guru, StackOverflow, and GeeksforGeeks played a pivotal role in improving our coding style and methodologies.

Statement on the OOAD process for your overall Semester Project

1. Problems with implementation of Observer pattern
    - The Observer pattern posed challenges in our project due to the dynamic nature of the data. Traditional Observer patterns assume a stable subject for notifying observers of changes, creating difficulties in handling highly dynamic data with frequent updates.
    - This led to potential performance bottlenecks and challenges in managing notification flows. As a solution, we transitioned the Observer pattern from notifications to logging, providing a more efficient way to capture and track dynamic data changes.

2. Problems with implementation of Factory pattern
    - Originally, our plan involved implementing the Factory pattern to facilitate the addition of menu items for any given restaurant. However, during the initial stages, we encountered challenges in identifying distinct ways in which various menu items would differ within the project's context
    - The lack of clear classification prompted a reevaluation of our approach, leading us to pivot towards the Strategy pattern. This shift allowed us to effectively implement a discount class for the existing menu items, providing a more adaptable and scalable solution for our project.

3. Readable project because of MVC Pattern
    - The MVC pattern significantly enhanced our project's readability by providing a clear separation of concerns. However, adapting to the dynamic nature of data required careful consideration. While MVC traditionally excels in managing structured request-response flows, our dynamic data posed challenges in maintaining clarity.
    - To address this, we prioritized documenting interactions between Model, View, and Controller components, ensuring a comprehensive understanding of the dynamic data's impact. This documentation, coupled with adherence to MVC principles, played a pivotal role in preserving readability as our project evolved.