TomoMacro (Macro nutrient tracker)

Luke Pan, Kyle Shi

Final State of System Statement

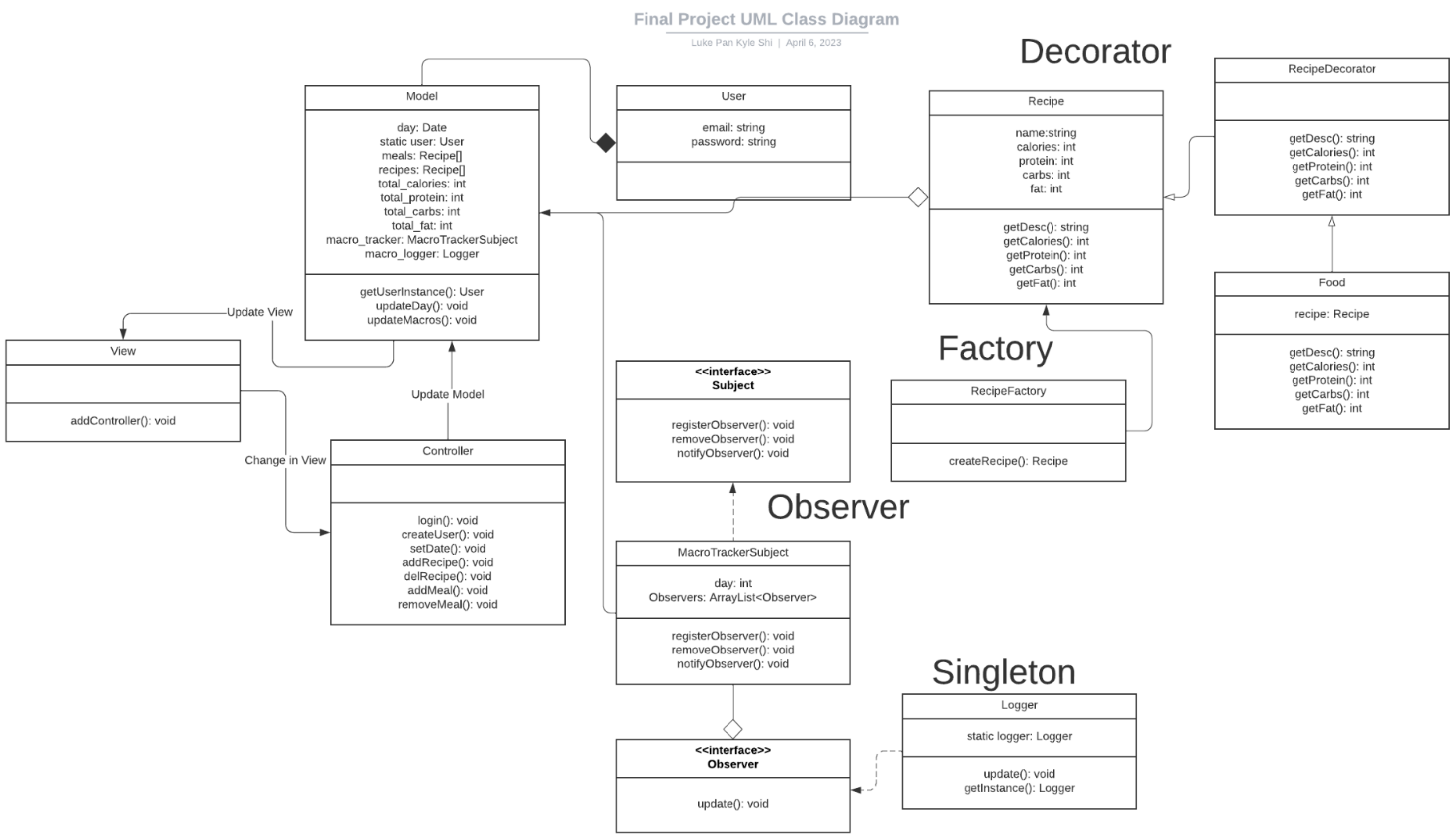
Below is a list of the requirements we listed out in Project 5 and the status of their completion.

- Publicly Hosted - **dropped**
- Login/Registration - **completed**
- Password Encryption - **completed**
- Adding/editing/deleting recipes - **completed**
- Adding/deleting foods - **partially complete**
- Calculating Macros - **completed**
- Interactive UI - **partially completed**
- Day Based Macro Tracking - **dropped**

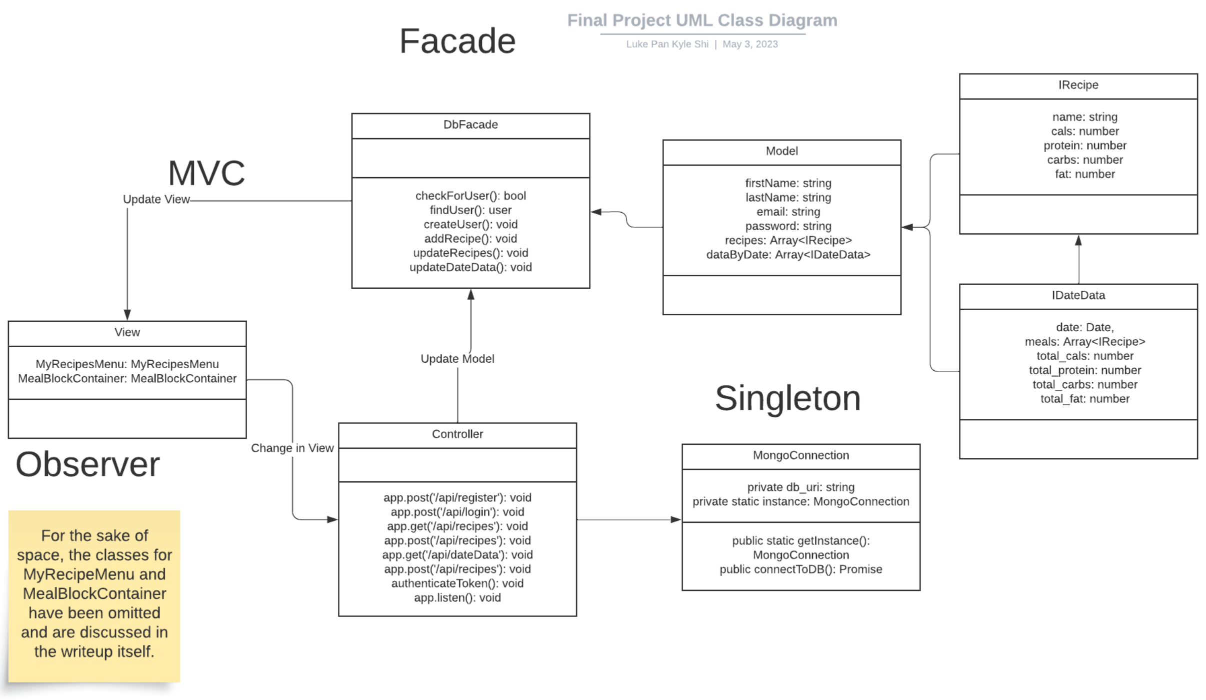
From projects 5 and 6, we implemented a large amount of the features we'd laid out for ourselves because in Project 6, we were still getting used to the new tech stack and setting everything up. We'll start by discussing the completed requirements. As for the login/registration and CRUD operations with recipes, they're both fully functional and all changes are reflected in the database. The password encryption is also completed and uses the Bcrypt hashing algorithm. Calculating macros works as well when users input the foods they've eaten. We were only able to partially complete the CRUD operations for foods and the UI as a lot of refactoring and headache would've been required to complete these two requirements; this is the same case with the Day Based Tracking - without finishing the CRUD operations for foods, we couldn't implement day based macro tracking. Finally, we decided that our web app wasn't complete enough to host so we did not complete this requirement.

Final Class Diagram and Comparison Statement:

Project 5 Class Diagram:



Final Class Diagram:


Final Project UML Class Diagram
Luke Pan Kyle Shi | May 3, 2023
Facade
MVC
Singleton
Observer
DbFacade
checkForUser(): bool
findUser(): user
createUser(): void
addRecipe(): void
updateRecipes(): void
updateDateData(): void
Model
firstName: string
lastName: string
email: string
password: string
recipes: Array<IRecipe>
dataByDate: Array<IDateData>
IRecipe
name: string
cals: number
protein: number
carbs: number
fat: number
IDateData
date: Date,
meals: Array<IRecipe>
total_cals: number
total_protein: number
total_carbs: number
total_fat: number
Update View
View
MyRecipesMenu: MyRecipesMenu
MealBlockContainer: MealBlockContainer
Update Model
Change in View
Controller
app.post('/api/register'): void
app.post('/api/login'): void
app.get('/api/recipes'): void
app.post('/api/recipes'): void
app.get('/api/dateData'): void
app.post('/api/recipes'): void
authenticateToken(): void
app.listen(): void
MongoConnection
private db_uri: string
private static instance: MongoConnection
public static getInstance(): MongoConnection
public connectToDB(): Promise
For the sake of space, the classes for MyRecipeMenu and MealBlockContainer have been omitted and are discussed in the writeup itself.

We stayed with most of the same design patterns that we decided on at the beginning of the project. However, due to learning more about the nature of the project through coding, we implemented them differently than originally planned.

The MVC pattern was implemented through using MongoDB as our Model, React for our View, and Express for our Controller. This follows the MVC pattern because all the interactions that the user tries to make with the data on the front end (CRUD operations w/ meals/recipes) through react, are sent to our controller (Express), which handles the data and sends it to our model (database) using Mongoose. When data is changed in the database, our view gets notified of this by fetching the database data and updates itself (components rerender).

React inherently implements the observer pattern through its rendering of components. To explain this, components have states (or attributes that change throughout the components lifetime), and whenever its state changes, the component rerenders on the screen to show the updated version of the component. All children of a parent component are eligible to change the state of the parent component, this is akin to the PubSub or the Subject Observer pattern. For our app specifically, whenever a user adds a food that they've eaten to a meal, the macro footer at the bottom of the app gets notified, or updated of this change and updates the total macro values that it displays (observer!!).

Also, instead of using the decorator pattern, we decided to use the Facade pattern as it better suited our project. Finally, we decided not to implement the Factory pattern as it did not make sense in the context of our project.

Third-Party code vs. Original code Statement:

The majority of the code was original, however small amounts of template code was taken from the following sources:
https://www.mongodb.com/languages/mern-stack-tutorial
https://www.youtube.com/watch?v=DZBGEVgL2eE
https://www.youtube.com/watch?v=mbsmsi7l3r4&t=639s

Statement on the OOAD process for your overall Semester Project:

1. Due to Javascript not being an object based language, it was difficult for us to find ways to implement design patterns. Most designs of web applications that we used as reference did not include any design patterns.
2. When looking for ways to implement design patterns, we decided that Facade fit our needs well. We used the Facade design pattern to create an interface between the server and the database. This interface was simpler and exposed a subset of all the possible database functions enabled by Mongoose. This allows for cleaner and more readable code that is easier to iterate on in the future.
3. We needed to connect to our database somehow, and we only wanted to connect to it once. Because of this, we decided to implement a Singleton class for our database connection. This was a pretty simple decision to make and was a very relevant use case for the singleton pattern.
4. We wanted an easy way to reflect the changes in a user's macros and needed a way to build a clean UI. So we decided to use React. React components use the observer pattern. This made it easy for us so whenever a food is added to a meal, the macro footer is notified/updated by the macro values of the food and reflects the changes.