SharePlate: Team 2 Final Report

Table of Contents

Solution functionalities

Section 1: Requirements specification

Business Driver

SharePlate is a platform for food providers to serve surplus food to customers at a low price or even provide food for free. It aims to address the growing issue of food waste and provide an alternative solution for businesses and consumers.

Main Features

- Identity and Access Management
  - I can register as a customer or as a shop, and I can log in with the corresponding identity. For different identities, the application page view will be different, which means certain behaviors not belonging to this identity will be constricted both in GUI and API calls.

- Food Posting
  - As a food provider, I can post food items for sharing or sale. A posted food should contain the food name, price per unit, amount available, its description in detail (optional), and the food image URL (optional).
  - For any food provided by me, I can modify the information mentioned above, or delete it from my shop.

- Placing Order
  - As a customer, I can place an order for the food items posted by one food provider at a time. I select the food I like together with the quantity I wish to get. There are three types of order to be chosen from: the basic order, order with togo-boxes and curbside pickup order. I can cancel any order created by me before it is completed, which means I can cancel the order when it's in Pending or Confirmed status.
  - As a food provider, I can proceed or cancel any orders related to my shop. If an order is placed or canceled, the inventory will be updated automatically.

- Order Review
  - As a customer, after the order is completed, I can rate the shop and compose a review about this order. The rating and review can be found on the shop's review page as well as the map-based search view.
  - As a food provider, after the order is completed, I can rate the customer and compose a review about this order. The rating and review can be found on the customer's main page.

- Shop Subscription and Notification push
  - As a customer, I can get a notification push when my order is completed or canceled. Furthermore, I can subscribe to shops I like and when there's a new food post, I will also get notification.

- Map & Tag-based Searching
  - As a customer, I can have a map view with all the shops close to my current location. All the shops will be displayed as circles with the average rating, and the unavailable shops will be colored gray.
  - As a customer, in the list view of shops, I can choose the search criteria, and further choose the sub-tags under that criteria to search for shops that meet the criteria.
  - As a customer, on the shop detail page, I can choose to show the direction to that shop. And I will be redirected to a third-party navigation page.

Section 2: Solution demo and installation guide

Demo

We will not include demo videos in this document.[1]
If you want to try SharePlate, please go to https://shareplate.pages.dev

Project Repositories

- Frontend: s23-t2-shareplate-frontend
- Backend: s23-t2-shareplate-backend

Before Installation - Environment Variables

Before the installation, please download the **`.env`** files and place them in the root directory of the frontend and backend repositories. These **.env files contain critical information** for the application to work properly. Make sure you have correctly placed them before you move to the next section.

- Frontend **`.env`** file: Download
- Backend **`.env`** file: Download

Installation Guide - Docker (Recommended)

In order to start up the services, you need to first install Docker on your machine. Please refer to this page to get the instructions about how to install Docker Desktop on Mac. If you encounter any issues with Docker on your machine, please use the *"Installation Guide - Local"* instead.

- Frontend

  1. Clone the repository
     `git clone https://github.com/cmusv-sasd/s23-t2-shareplate-frontend.git`
  2. Enter the directory
     `cd s23-t2-shareplate-frontend`
  3. Make sure Docker is installed and running on your machine
  4. Start the Docker container
     `docker compose up --build`
  5. You can visit SharePlate by going to this URL
     `http://localhost:4173/`

- Backend

  1. Clone the repository
     `git clone https://github.com/cmusv-sasd/s23-t2-shareplate-backend.git`
  2. Enter the directory
     `cd s23-t2-shareplate-backend`
  3. Make sure Docker is installed and running on your machine
  4. Start the Docker container

[1] We have already demonstrated the application during the demo session. Professor Rafal has given us verbal permission to skip the demo section in this document.

`docker compose up --build`

5. You can check if the backend service is running by accessing this url
   `http://localhost:3000/api/healthcheck`

Installation Guide - Local

- Frontend

1. Clone the repository
   `git clone https://github.com/cmusv-sasd/s23-t2-shareplate-frontend.git`
2. Enter the directory
   `cd s23-t2-shareplate-frontend`
3. Install dependencies and start the development server
   `yarn && yarn dev`
4. You can visit SharePlate by going to this URL
   `http://localhost:5173/`

- Backend

1. Clone the repository
   `git clone https://github.com/cmusv-sasd/s23-t2-shareplate-backend.git`
2. Enter the directory
   `cd s23-t2-shareplate-backend`
3. Install dependencies and start the development server
   `npm i && npm start`
4. You can check if the backend service is running by accessing this URL
   http://localhost:3000/api/healthcheck

Solution architecture & design

Section 3: System architecture

Here we present a final version of our architectural diagram. We use MVC structure to divide our codebase into front-end views and back-end models and controllers. In order to make our diagram visually clear, we use green components to represent front-end pages and back-end operations for customers, blue components for vendors, and the combination of two colors to represent components for both sides.

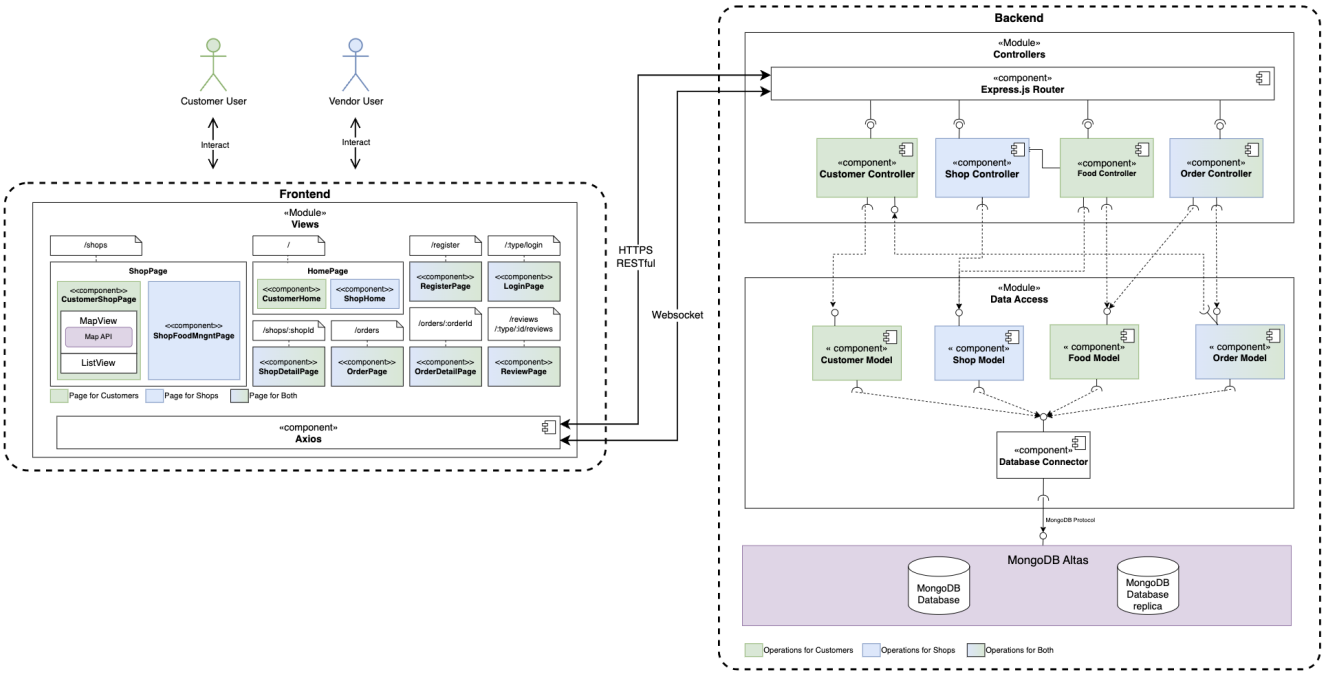

Figure 3.1: Architecture diagram of SharePlate

Section 4: Design decisions

1. MVC structure

- Cost-Effective for project scope

  Compared to more complex designs such as Microservices or Service-Oriented Architecture (SOA), MVC architecture is simpler and more cost-effective. It allows for efficient development, testing, and maintenance of the application. This is particularly advantageous for a project like SharePlate, which is relatively straightforward, and doesn't require the additional complexity and costs associated with microservices or SOA.

  Unlike other complex architectures that may require the definition of many interfaces, MVC is relatively simple. This simply means that there's no need for over-defining interfaces which can increase unnecessary overhead. By reducing such overhead, the MVC architecture ensures smooth and efficient operation of the application, thereby improving its performance.

- Reusability

  By implementing the MVC architecture in the SharePlate application, we're able to employ certain strategies to uphold our quality attributes. Firstly, we can best optimize the use of object-oriented design in this project scope within each layer to maximize reusability and minimize code duplication. Secondly, we effectively utilize the utilities and components defined within each layer to enhance cohesion. This approach helps to create a well-structured and maintainable codebase.

  - Model Layer

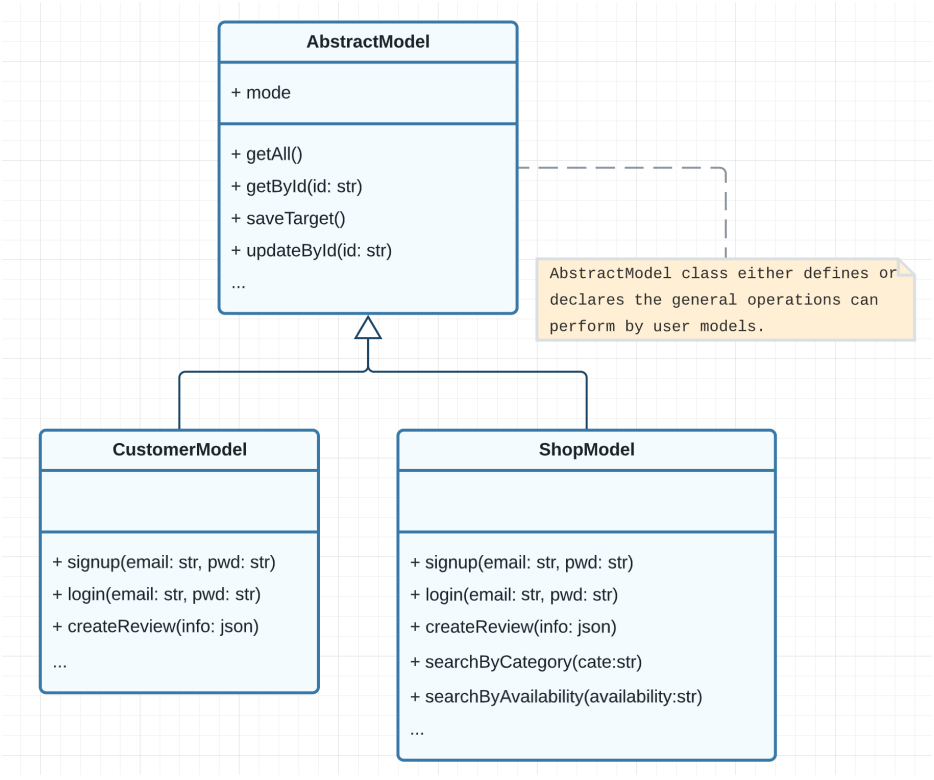

Figure 4.1: Class diagram of AbstractModel class

As the SharePlate application applies MVC architecture, we are able to incorporate the super class to define the general operations for all concrete implementations in the same layer. For example, in the model layer, we decide to declare an AbstractModel in order to increase reusability and reduce code duplication in the CustomerModel class and ShopModel class.

This **`AbstractModel`** class in the MVC architecture backend model layer enhances reusability by providing a general set of operations for interacting with the database, which needed subclasses can inherit. The class encapsulates common operations such as getting a record by email or id, retrieving all data, and saving data. This means that every model that extends this abstract class can reuse these general methods without having to redefine them, preventing the code duplication.

It also allows for flexibility with different database types through the DBs constructor parameter and setMode method, enabling the switch between different database modes such as production mode or mock mode if needed. By following this approach, code duplication is significantly reduced, enhancing the readability of the system.

Source Code: `s23-t2-shareplate-backend/app/models/AbstractModel.js`

- Controller layer

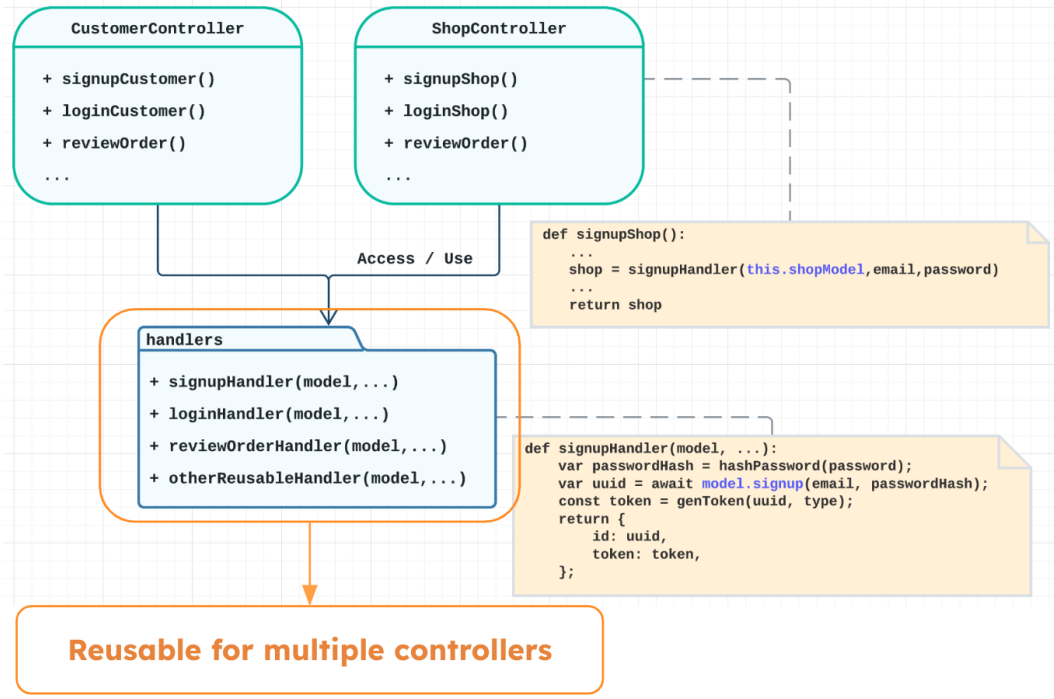

Figure 4.2: Conceptual illustration for reusable utilities

Our system employs the Model-View-Controller (MVC) architecture, which inherently supports reusability, thereby enhancing the efficiency of our development process. As a part of this setup, we've created reusable components, including handler classes and middleware.

These components are defined in the controller layer as common operations, allowing multiple controllers to access and reuse the same utilities, which contributes to code efficiency and maintainability.

More specifically in the utilities we defined in the controller layer, a key feature we can design these utilities is that they are agnostic about the specific models conducting certain tasks, adding to their versatility. For instance, our reusable login and signup handler can be employed across different controllers such as the **`shopController`** and **`customerController`**. This design allows us to reuse critical code snippets across the application, reducing redundancy and ensuring consistency in our operations.

Source Code: `s23-t2-shareplate-backend/app/handlers/*`

2. MongoDB Atlas

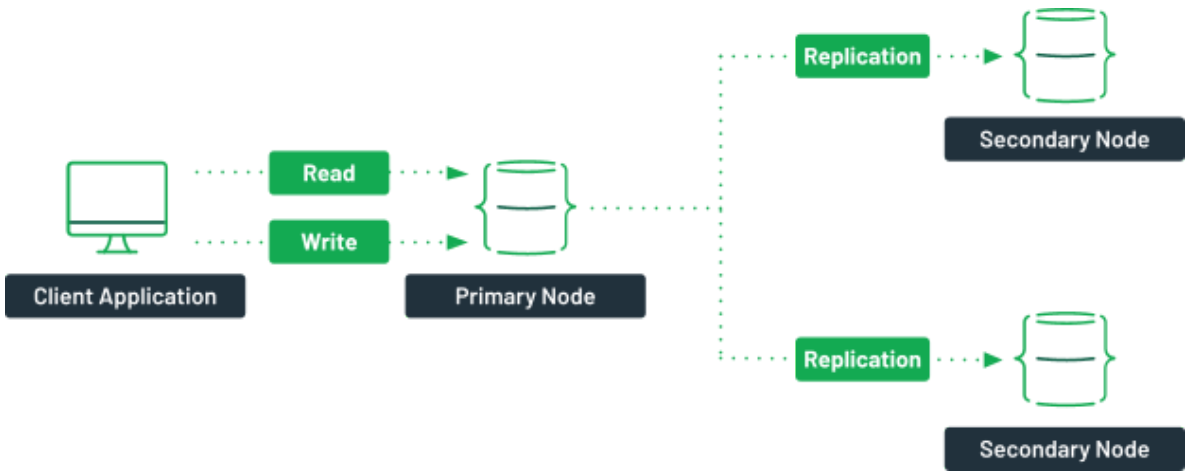

Figure 4.3: Atlas replication strategy to guarantee availability

MongoDB Atlas is a fully managed, global cloud database service provided by MongoDB Inc., it offers a Database as a Service (DBaaS) solution that allows us to deploy, manage, and scale our MongoDB databases in the cloud. It has a few key features which can greatly contribute to the availability of our application:

- High Availability: Atlas deploys your database across multiple availability zones within a region, providing automatic failover and redundancy in case of hardware or network failures. It offers a 99.95% uptime Service Level Agreement (SLA) for dedicated clusters. This means that MongoDB commits to having your database available at least 99.95% of the time.
- Automatic Scaling: Atlas allows you to scale your database vertically and horizontally on-demand or automatically based on the resource usage and performance metrics of your cluster.
- Managed Backups: Atlas takes care of automated backups, point-in-time recovery, and snapshot retention, ensuring that your data is protected and can be restored in case of any issues or data loss.

To sum up, MongoDB Atlas can make sure the DBs are always at service and are responsible at any time and at any place, even when the I/O traffic is experiencing a high volume, which further guarantees our availability of NFRs.

Source Code: There is no source code involved in this design decision. Following are the images related to Atlas deployment for your reference.

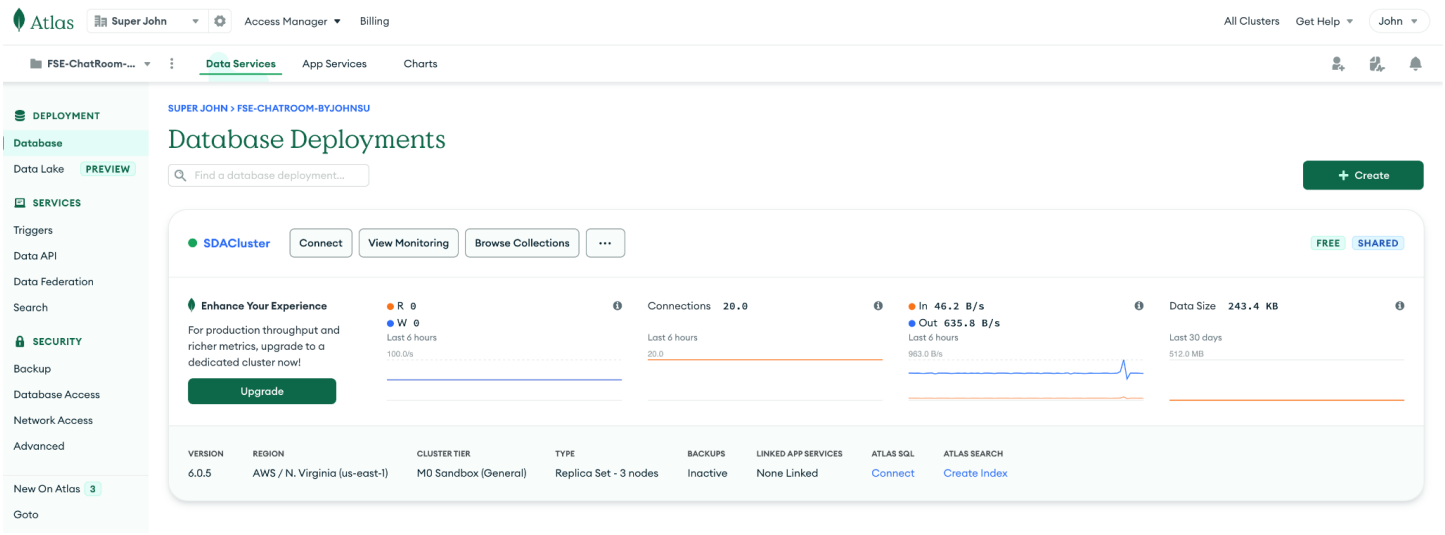

Figure 4.4: Database Deployments at MongoDB Atlas

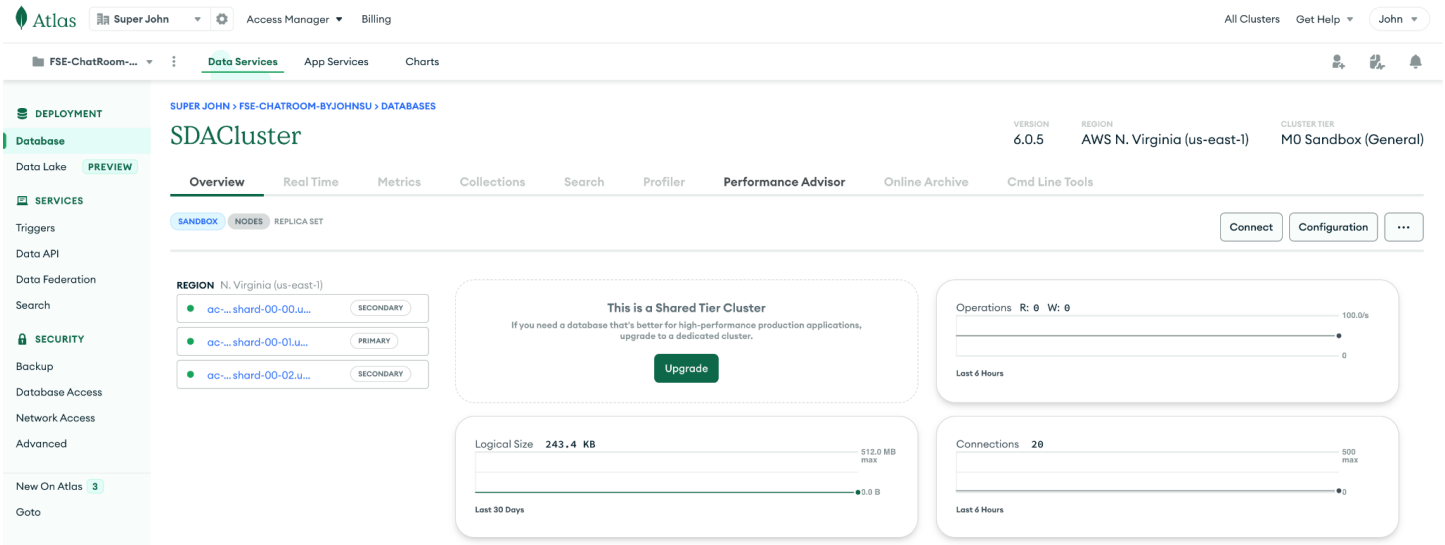

Figure 4.5: Cluster performance overview

3. Reusable Frontend Pages and Components

In the SharePlate project, reusability is considered a crucial aspect of software development. To approach better reusability, one of the design decisions was to implement reusable frontend pages using React Router. By using React Router, we can create adaptable pages that can be easily customized for different types of users, reducing the need for code duplication.

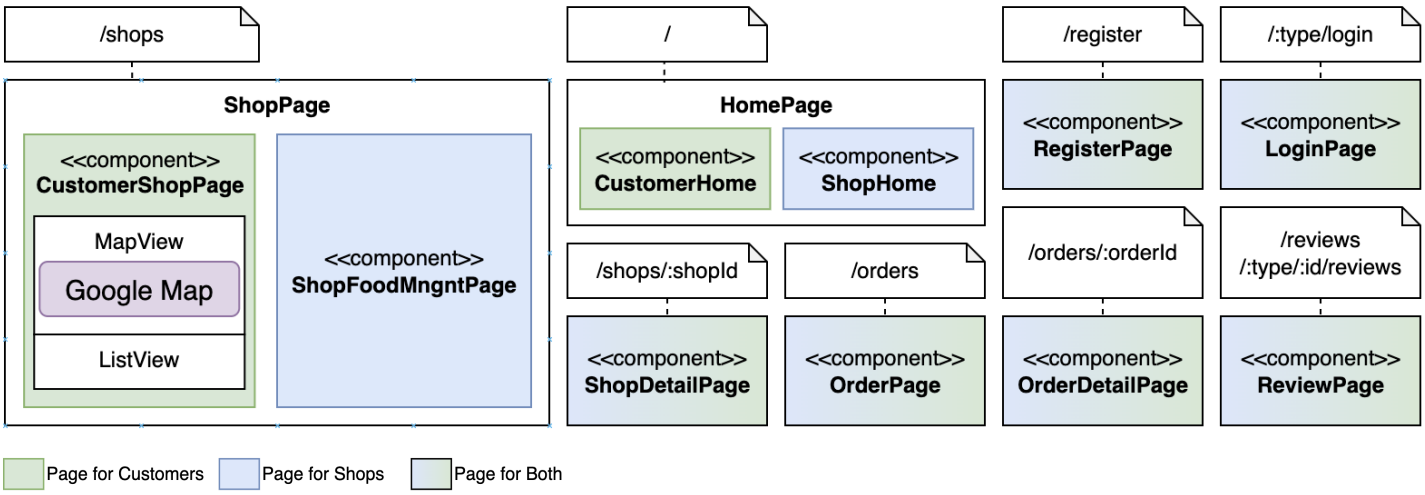

Figure 4.6: Reusable Frontend pages

For instance, the home page for both the customer and shop pages can be routed to the same endpoint ("/"), while we also implemented a strategy to render different home pages for shops (**ShopHome**) and customers (**CustomerHome**). This approach significantly enhances the reusability of the codebase and reduces the amount of code needed to maintain different pages for different users

Another key feature of the SharePlate frontend is the use of React Components, which helps in preventing code duplication, making the code more maintainable and scalable. By using reusable components, we can create a consistent and standardized look and feel throughout the application, without the need to copy-paste code all the time. This approach enhances the reusability of the codebase and allows for easier management of the code.

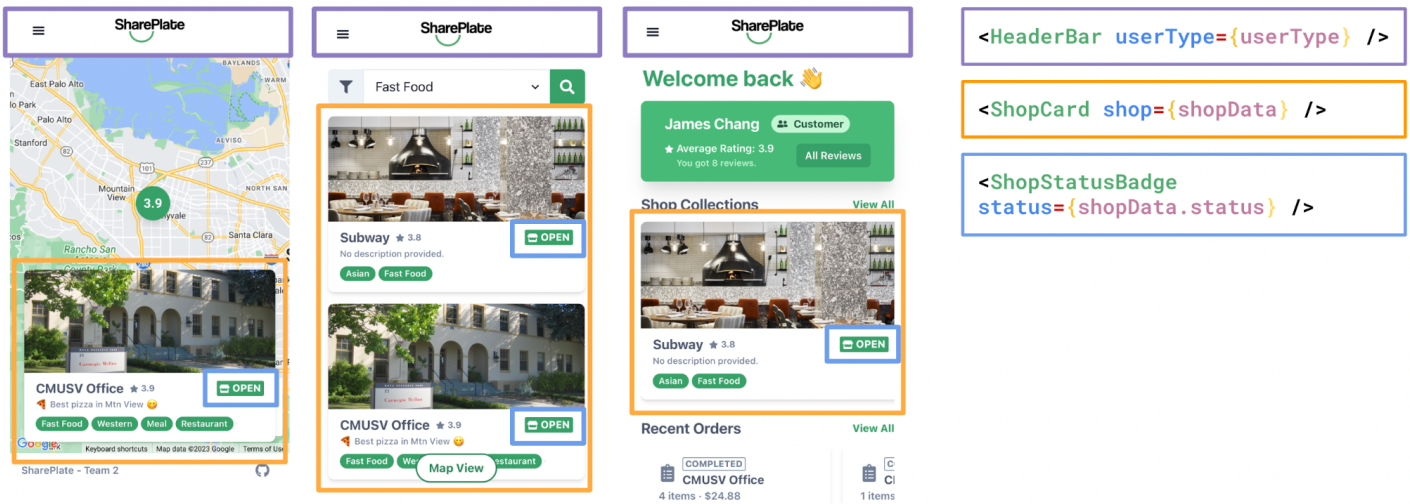

Figure 4.7: Frontend reusable component design

For instance, we use a **`HeaderBar`** component that can be used to display a navigation bar on different pages of the application. Similarly, we use a **`ShopCard`** component to display shop-related information and a **`ShopStatusBadge`** component to display the status of a shop. By using these components, we can easily manage our code, remove duplicates and improve maintainability.

To sum up, the use of reusable frontend pages and components enhances the reusability of the codebase, making the software development process more efficient and effective. It fosters the achievement of quality attributes such as maintainability, scalability, and testability. The consistency and standardization achieved by using reusable components contribute to the maintainability and scalability of the application, while the reduced amount of code duplication enhances the testability of the software.

Source Code:

- Reusable Pages: `s23-t2-shareplate-frontend/src/pages/*`
- Reusable Components: `s23-t2-shareplate-frontend/src/components/*`

4. Customized Middleware Handlers

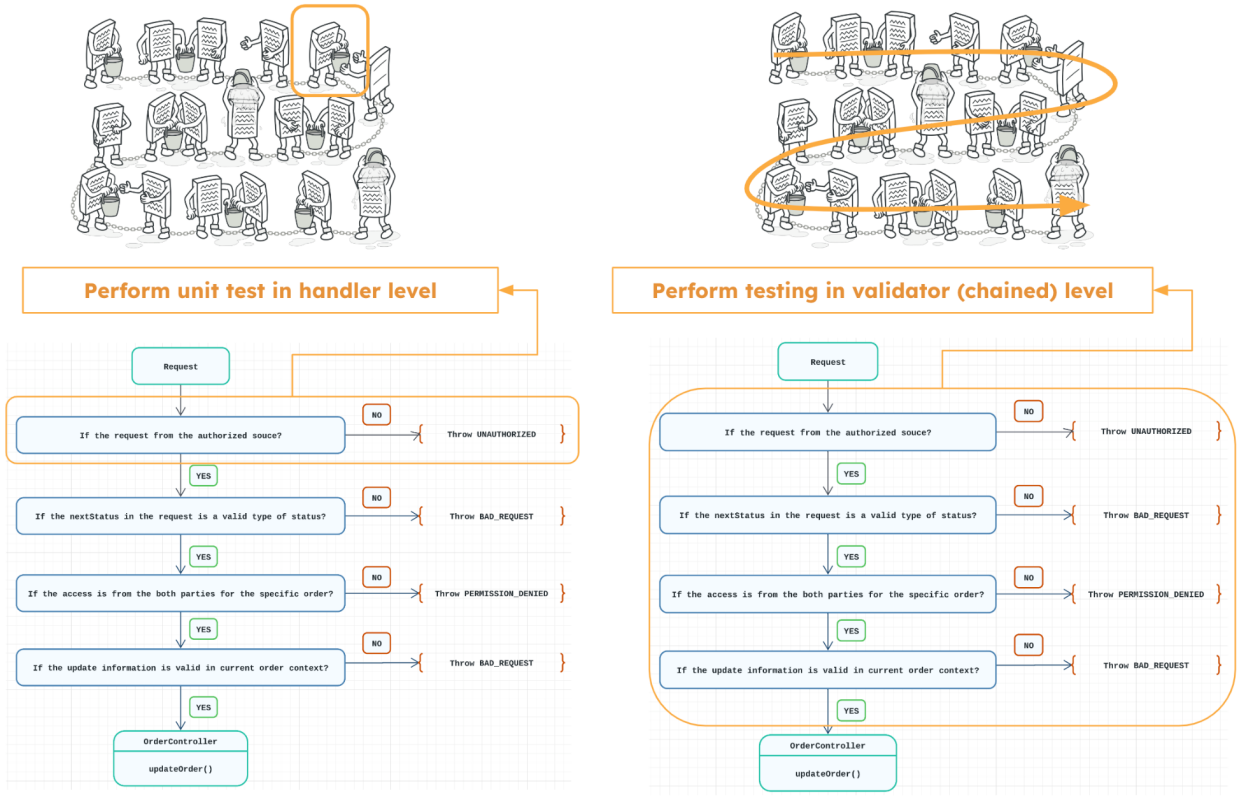

Figure 4.8: Conceptual illustrations for different testing granularity

Choosing to self-implement the handlers and validators, as opposed to relying on the original middlewares provided by the express module, can significantly enhance our testability and the overall quality of our software. By implementing our own chain, we have the flexibility to test validation behaviors in a more granular and controlled manner.

For instance, we can individually test each handler to ascertain whether it verifies requests as expected, ensuring the accuracy and reliability of each component in the chain. Moreover, we can also perform unit tests at the chain level, verifying that the entire chain of handlers validates requests correctly when working together.

Source Code

- Implementation: `s23-t2-shareplate-backend/app/middlewares/*`
- Unit test: `s23-t2-shareplate-backend/test/handler.test.js`

5. Frontend Deployment - Cloudflare Pages

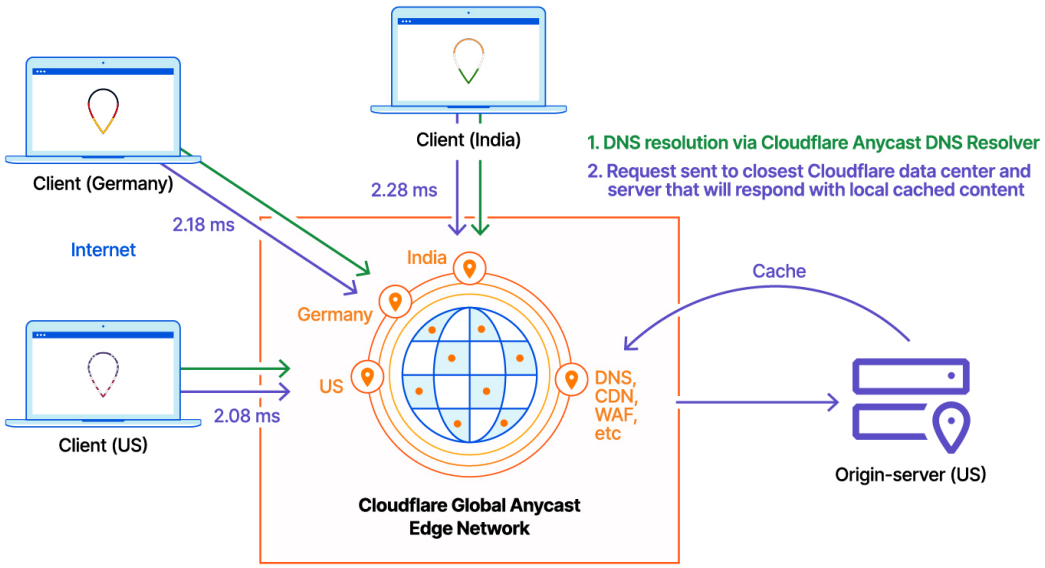

Figure 4.9: Cloudflare's CDN functionality

Cloudflare Pages is a free site hosting service that enhances SharePlate's availability because of these features that it offers:

1) **Content Delivery Network (CDN)**
   One of Cloudflare Pages' core features is the content delivery network (CDN), which stores cached versions of SharePlate frontend on servers located all around the world. This means that when a user requests a website, Cloudflare can serve the cached version from the nearest server, reducing latency and improving website load times. Improved website load times are a critical factor in availability. When a website is slow to load, users are more likely to abandon it, leading to decreased availability. By reducing latency through its CDN, Cloudflare Pages can help ensure that SharePlate is always available and accessible to users.

2) **DDoS Protection**

Cloudflare offers a range of security features that can further enhance availability. For example, it includes DDoS protection, which helps prevent malicious attacks that could disrupt our frontend service and make it unavailable.

Overall, Cloudflare Pages and its CDN functionality can help improve website availability by reducing latency, enhancing security, and protecting against downtime caused by malicious attacks.

Source Code

There is no source code involved in this design decision. Following are the images related to frontend deployment for your reference.

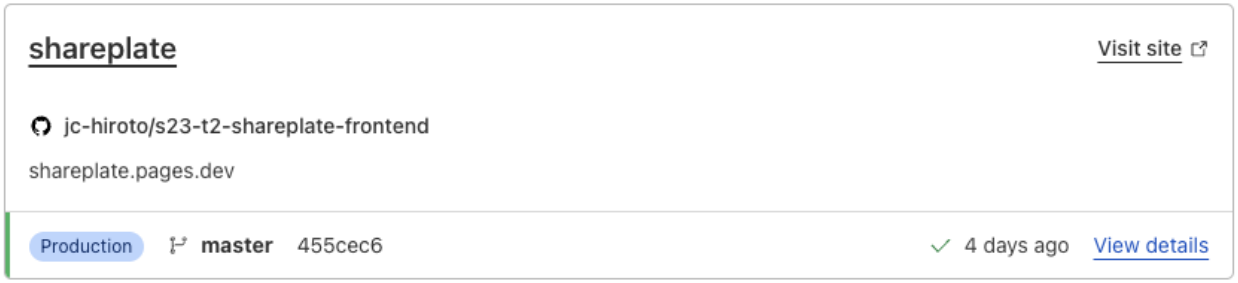

Figure 4.10: Cloudflare Pages Deployment Information

6. Backend Deployment - Azure App Service

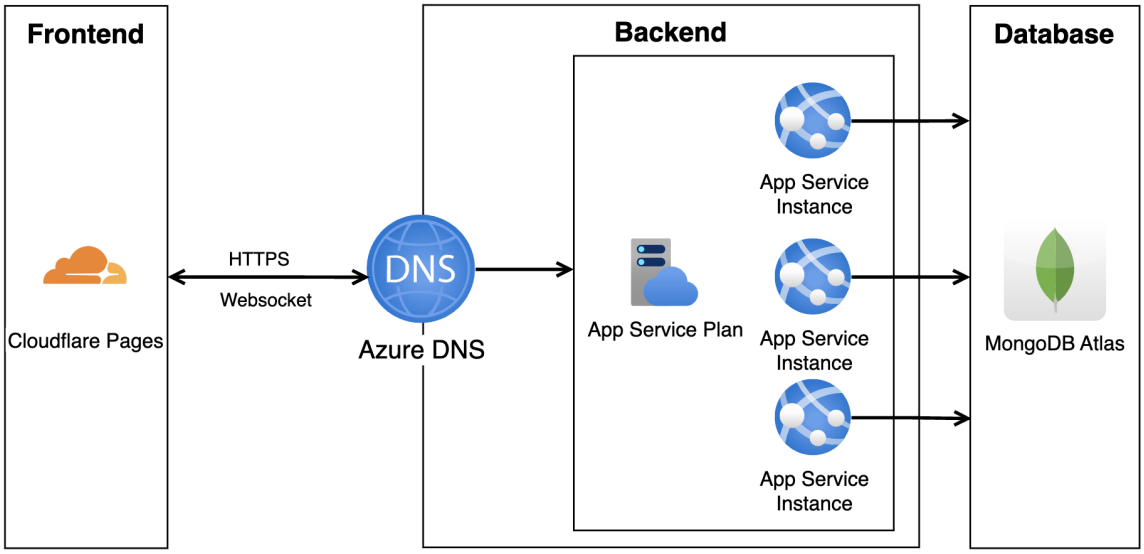

Figure 4.11: SharePlate's Backend "Multi-instance" Deployment Diagram (Simplified)

Azure App Service is a platform-as-a-service (PaaS) offering from Microsoft Azure that enables developers to deploy and run web applications and backend services on the cloud. Deploying our backend services to Azure App Service supports SharePlate's availability because of the following reasons:

1) **Increased Redundancy**
   By deploying SharePlate's backend service on three separate instances, we increase redundancy in our system. In case one instance fails or becomes unavailable, the other two instances can continue to serve requests, minimizing downtime and ensuring high availability.

2) **Load Balancing**
   Azure App Service automatically distributes incoming traffic evenly across all the instances, allowing for better load balancing. This helps prevent any single instance from being overwhelmed with too much traffic, which could result in slow response times or even downtime.

3) **High Availability SLAs**
   Azure App Service also offers high availability Service Level Agreements (SLAs), which guarantee a certain level of uptime and availability for our backend service. This means that Microsoft is committed to ensuring that our service is available and accessible to users, and will take necessary actions in case of any disruptions.

Source Code

There is no source code involved in this design decision. Following are the images related to backend deployment for your reference.

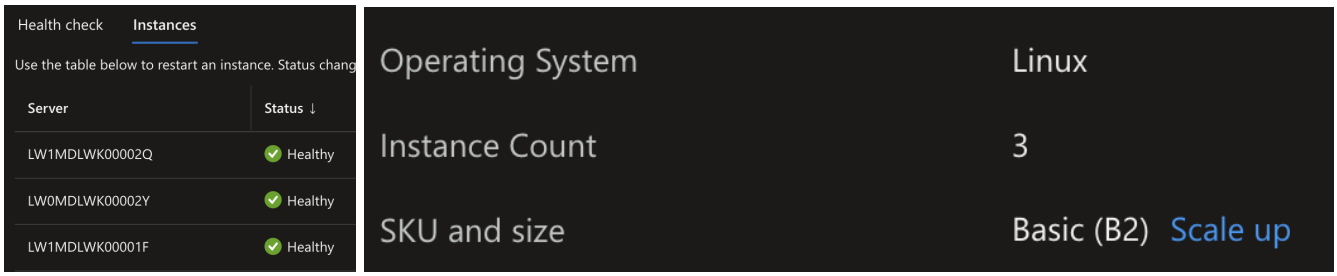

Figure 4.12: Azure App Service Deployment - Instance count and health status.

Section 5: Design patterns

1. Factory Method

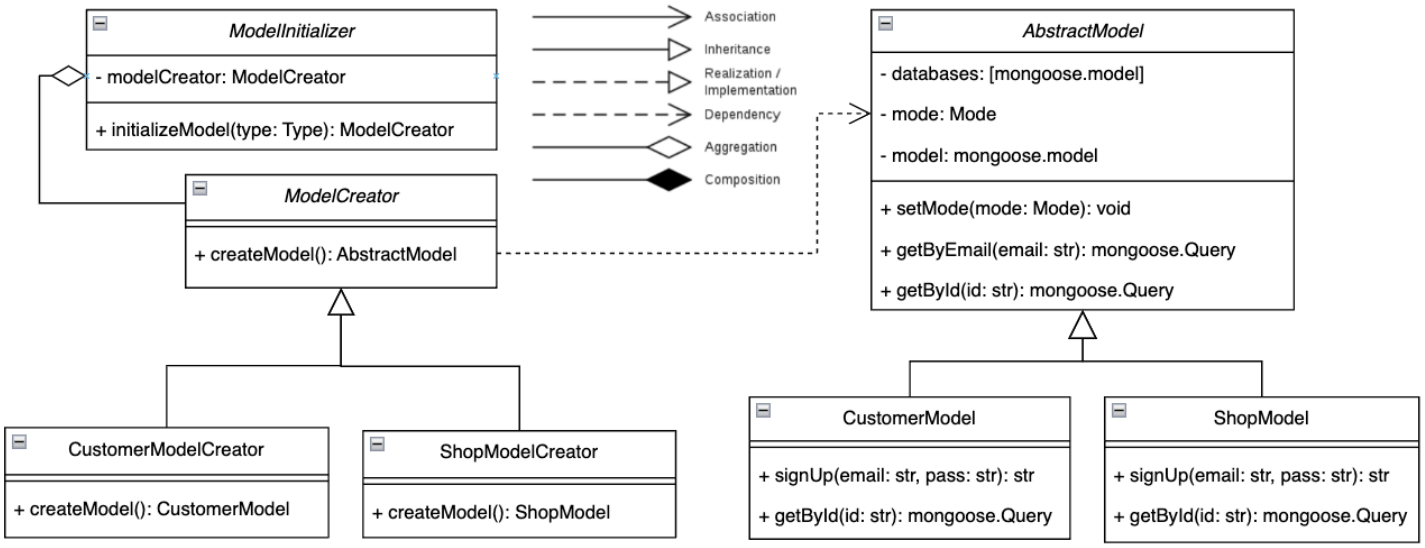

Figure 5.1: Class diagram of Factory Method in SharePlate

Problem / Rational

as two types of users: Customers and Shops. Customers acquire food from Shops, which are stored in separate MongoDB collections with different schemas. We require a unified interface or abstraction for common methods, including `signUp()` and `getById()`, between these two user types. Essentially, we are seeking a solution to create different types of user data access objects (Models) that share the same interface and abstraction.

Benefits

- Single Responsibility Principle: All of the model creation code is all in one place, which makes the code easier to maintain.
- Open-Closed Principle: Let us handle different types of users without modifying the controllers or bundle the logic inside the controller.

Implementation

Initially, the `ModelInitializer` will call different `ModelCreators` to create data models for the app to utilize. There are two model creators `CustomerModelCreator` and `ShopModelCreator`, which are both inherited from `ModelCreator`. Products created by the creators are `CustomerModel` and `ShopModel`, which are inherited from the abstract class `AbstractModel`.[2] There are some common methods declared and implemented at the abstract class level and its children can override certain methods if they need to make the concrete implementation user-type-specific.

Source Code

This pattern is implemented in `s23-t2-shareplate-backend/app/models`

- Model Initializer
- Creators:

[2] The concept of interface is only introduced in TypeScript but not Javascript. Thus, the `AbstractModel` here is implemented as an abstract class instead of an interface.

    - `creators/modelCreator.js`
    - `creators/customerModelCreator.js`
    - `creators/shopModelCreator.js`
- Model Abstract Class (Interface): `AbstractModel.js`
    - `customerModel.js`
    - `shopModel.js`

2. Chain of Responsibility

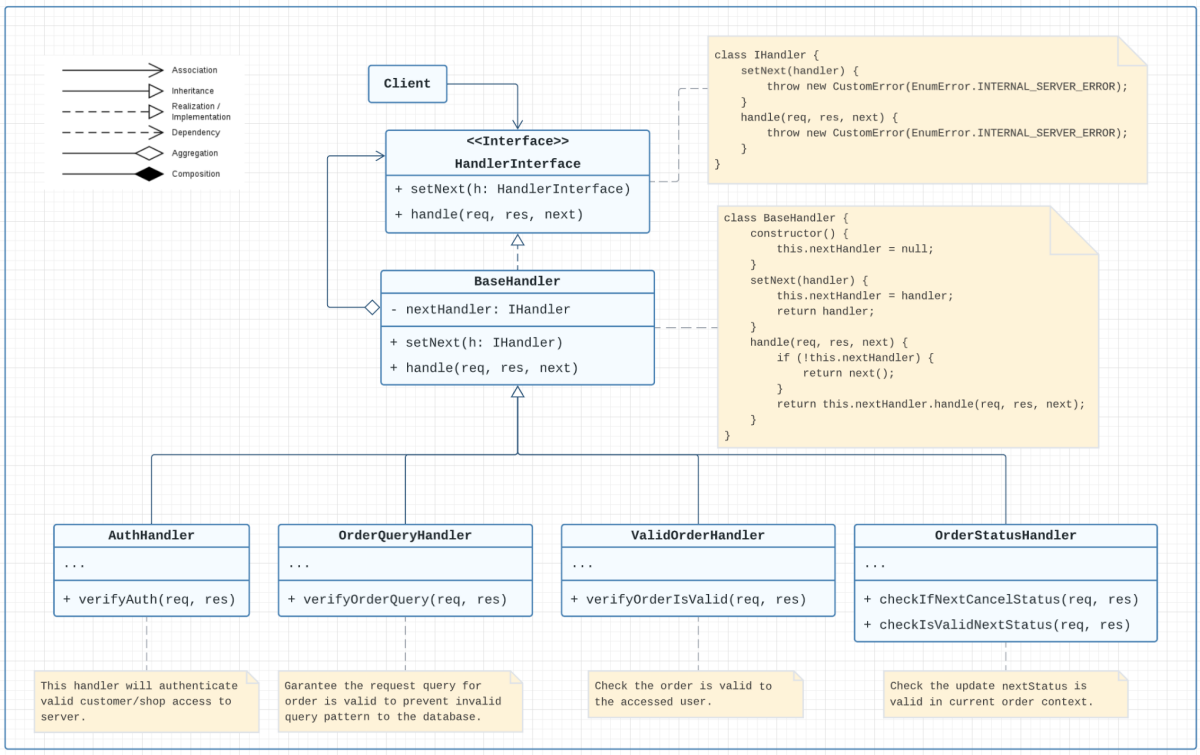

Figure 5.2: Class diagram of Chain of Responsibility in SharePlate

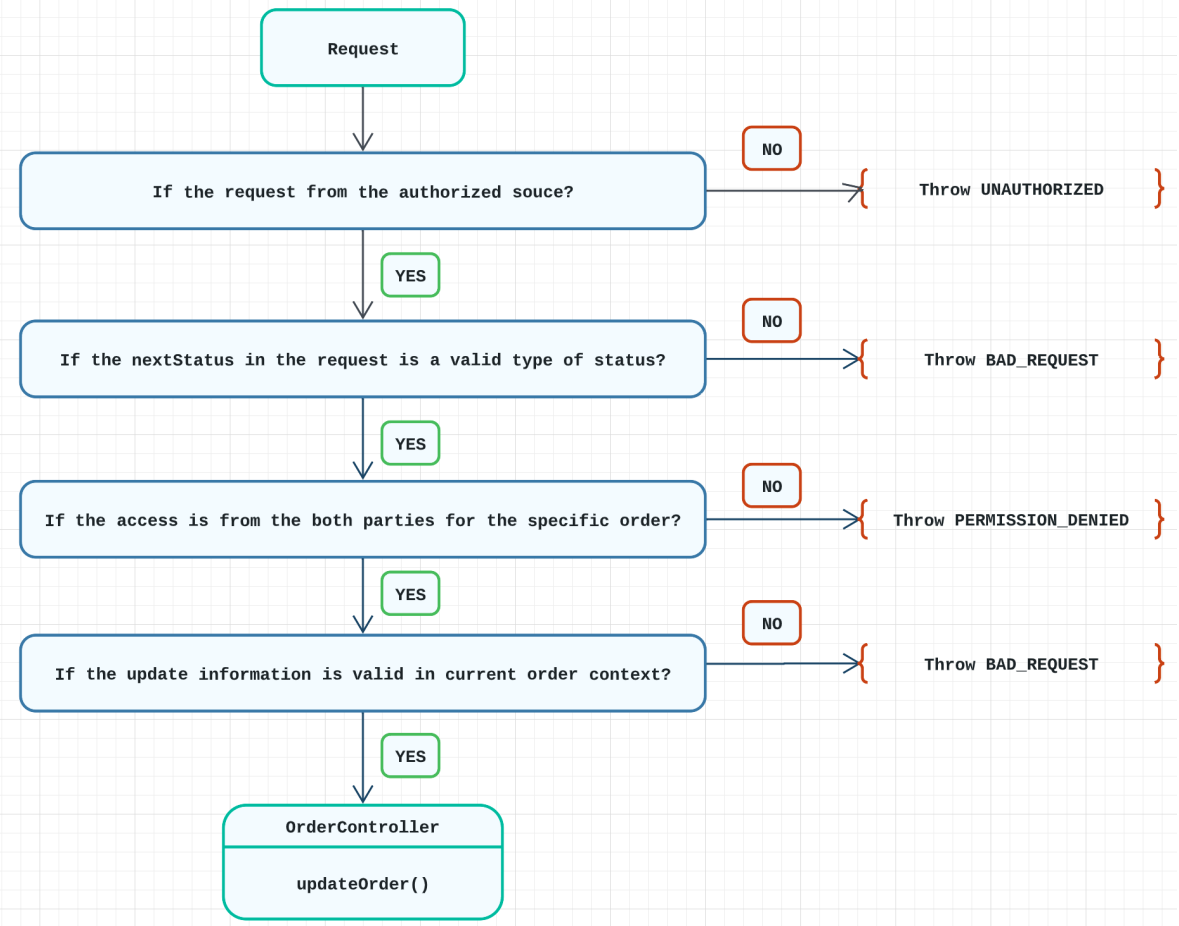

Figure 5.3: Visualization for the chain of Responsibility in update order part

Problem / Rational

During the development process, the SharePlate application faces the challenge of managing a complex set of validations and authorizations before creating or modifying orders. Also, as input requests might come from different types of users (Customer / Shop), the validations and authorizations behaviors will get much more complicated . To handle such a complex set of validations and authorization behaviors, our application adopts the Chain of Responsibility (CoR) design pattern, which allows the request to be passed through a chain of handlers until it is processed by the appropriate handler. CoR can be an efficient way to manage such complex processes by decoupling the processing logic into separate handlers and allowing each handler to be responsible for a particular type of validation or authorization.

Benefits

- Single Responsibility Principle: Each handler should handle a single validation or authorization behavior without handling functionality that does not belong to itself.
- Open-Closed Principle: Our system can easily extend additional checking behaviors without changing the original code, we can implement a new handler and chain it into the validator with ease.

Implementation

Initially, the **`baseHandler`** class (IHandler) is implemented to specify the **`setNext()`** and **`handle()`** operations for all the concrete handler classes. This allows us to create various types of handlers to execute specific actions as required. Furthermore, the **`validatorBuilder`** class is also implemented as a base class for linking the handlers in the required sequence. The validatorBuilder class defines general operations, including **`add()`**, **`getChainedHandlers()`**, and **`build()`** operations for clients to perform

chaining. In this way, adding additional handlers and connecting them as needed is a straightforward process. Additionally, this approach facilitates testing flexibility.

Source Code

This pattern is implemented in `s23-t2-shareplate-backend/app/middlewares`

- BaseHandler
  - `middlewares/handlers/baseHandler.js`
- `Other handlers`
  - `middlewares/handlers/*Handler.js (such as authHandler.js)`
- ValidatorBuilder:
  - `middlewares/validatorBuilder.js`
- `Other validators`
  - `middlewares/*Validator.js (such as crossTypeReviewValidator.js)`

3. Observer

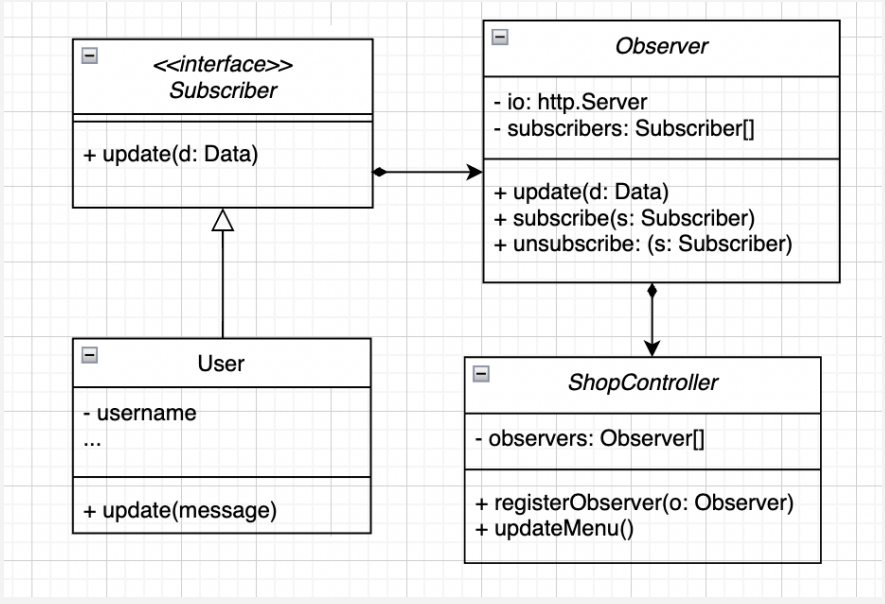

Figure 5.4: Class diagram of Observer design pattern

The Observer pattern is a behavioral design pattern that facilitates one-to-many communication between objects, where one object (subject) maintains a list of its dependents (observers) and notifies them automatically of any changes to its state. This report will discuss the implementation of the Observer pattern in the Shop subscription feature of the SharePlate backend.

Problem / Rational

The Shop subscription feature of the SharePlate backend requires real-time updates for users who have subscribed to their favorite shops. However, continuously polling the shop's database for changes is not feasible and can lead to performance issues. Moreover, managing and notifying a large number of subscribers when updates occur can be challenging.

Solution

The Observer pattern can be used to address the above problems. In this scenario, the Shop subscription feature uses the Observer pattern to allow customers to subscribe to their favorite shops and receive

real-time updates for any changes that occur. The customers act as concrete subscribers (Users), while the ShopMenuObserver acts as a specific observer that is responsible for notifying users when a shop's menu changes. The ShopController manages the Observer pattern for the Shop subscription feature, and it registers observers and notifies them when changes occur.

Implementation

The ShopMenuObserver is implemented using the following code:

```
class ShopMenuObserver {
    constructor(io) {
        this.io = io;
    }

    update(subscribers, data) {
        for (let i = 0; i < subscribers.length; i++) {
            const subscriberId = subscribers[i];
            this.io.to(subscriberId.toString()).emit("menuChanged", { data });
        }
    }
}

export default ShopMenuObserver;
```

Figure 5.5: Code implementation of ShopMenuObserver

The **`ShopMenuObserver`** contains an instance of the HTTP server and uses Socket.io to notify subscribers of changes to the shop's menu. On the other hand, the **`ShopController`** manages the Observer pattern for the Shop subscription feature and is responsible for registering observers and notifying them when changes occur. The code for the **`ShopMenuObserver`** and **`ShopController`** can be found in **`observers/ShopMenuObserver.js`** and **`controllers/shopController.js`**, respectively. The customers, who act as subscribers, are represented by the **`customerModel.js`** file.

Benefits

Using the Observer pattern simplifies the codebase by reducing the amount of code needed to manage subscriptions and notifications. It also improves the performance of the application by reducing the amount of unnecessary polling that takes place. Additionally, it makes it easier to manage and notify a large number of user subscribers when updates occur, making the feature more scalable.

Source Code

This pattern is implemented in **`s23-t2-shareplate-backend/app`**

- Observers:
  - `observers/ShopMenuObserver.js`
- Controllers:
  - `controllers/shopController.js`
- Subscribers:
  - `customerModel.js`

The implementation of the Observer pattern in the Shop subscription feature of the SharePlate backend provides an efficient solution to the problem of real-time updates for subscribers. This pattern simplifies the codebase, improves performance, and makes it easier to manage and notify a large number of subscribers. The Observer pattern is a powerful tool that can be used to achieve similar functionality in other scenarios where one-to-many communication is required.

4. Strategy

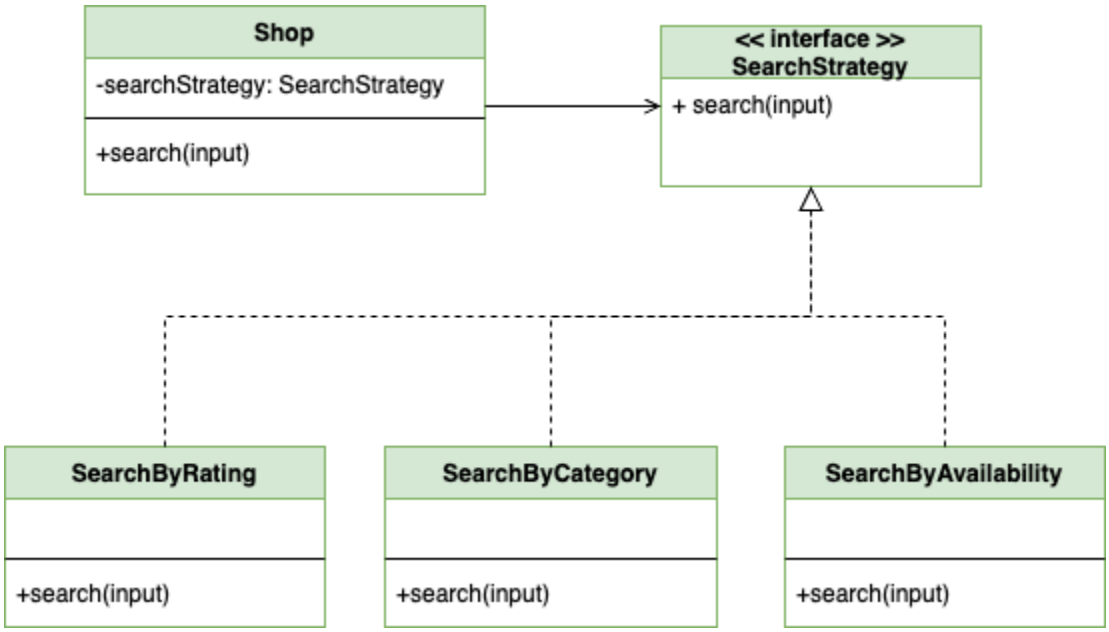

Figure 5.6: Original Search Methods Proposed

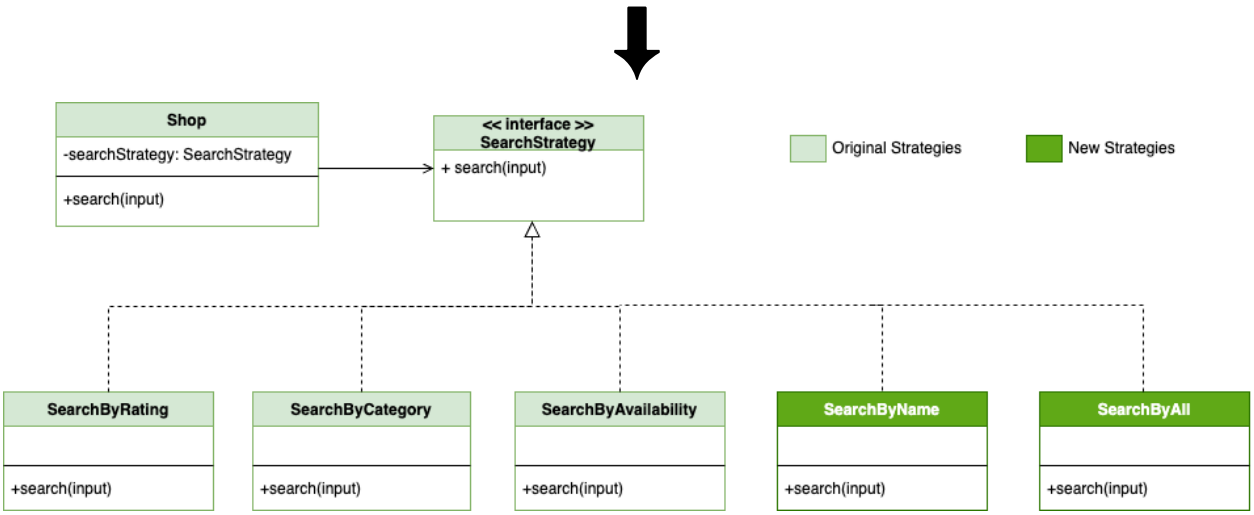

Figure 5.7: Updated Search Methods

Problem / Rational

Our application provides search functionality for customers to look for shops with specified conditions. We want to provide different kinds of search algorithms; for example, search by ratings to show good restaurants, or search by categories to show specific types of restaurants. At the same time, we want to keep our code structure clean and easy to maintain in the future. We are aware of the possibility that we need to add more search methods in the future. In order to keep the flexibility, we introduce a Strategy design pattern to our search feature.

Benefits

There are two major benefits for Strategy design pattern:

- Open/Close principle, provides flexibility to add new search algorithms
- The ability to switch between different search algorithms at runtime, which allows the customer to choose search criteria while using the application to look for shops

Initially, we proposed three different search algorithms, **`searchByRating`**, **`searchByCategories`**, and **`searchByAvailability`**. During our development process, we realized that we need two more search algorithms **`searchByName`** and **`searchByAll`**, and these two new search methods are very useful in practical situations. Because we implemented the Strategy design pattern in the beginning, it is very easy to add these new methods without adding additional burden to our code.

Implementation / Source Code

Our **`ShopController`** has a field **`searchStrategy`**, which is used to indicate the strategy method being used during runtime (Figure 5.8). We keep all different search methods in the strategy folder (Figure 5.9), **`s23-t2-shareplate-backend/app/strategies/searchBy{Condition}.js`**. During runtime, we set the searchStrategy field using different search methods, allowing customers to switch their search method at any time.

```
class ShopController {
    constructor(io, shopModel, orderModel) {
        this.io = io;
        this.shopModel = shopModel;
        this.orderModel = orderModel;
        this.profileDataHandler = new ProfileDataHandler();
        this.emailPasswordValidator = new EmailPasswordValidator().build();
        this.selfUpdateValidator = new SelfUpdateValidator().build();
        this.authValidator = new AuthValidator().build();
        this.crossTypeReviewValidator = new CrossTypeReviewValidator(this.shopModel).build();
        this.router = express.Router();
        this.handleRequests(this.router);
        this.searchStrategy = undefined;
        this.observers = [];
        this.registerObserver(new ShopMenuObserver(io));
    }
```

Figure 5.8: searchStrategy field in ShopController

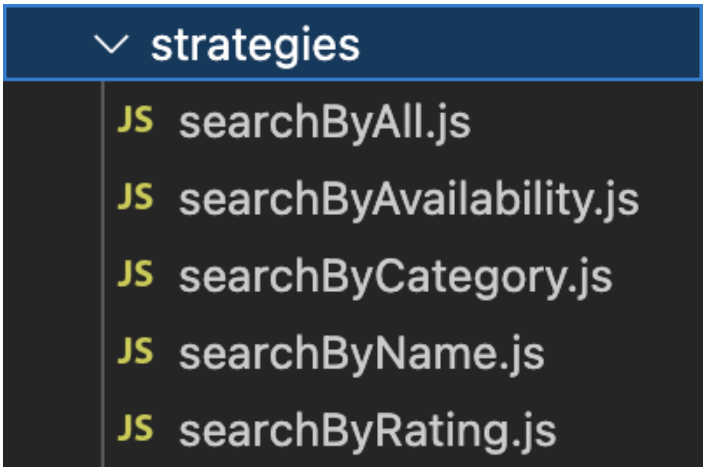

Figure 5.9: Strategies folder with different search algorithms

5. Decorator

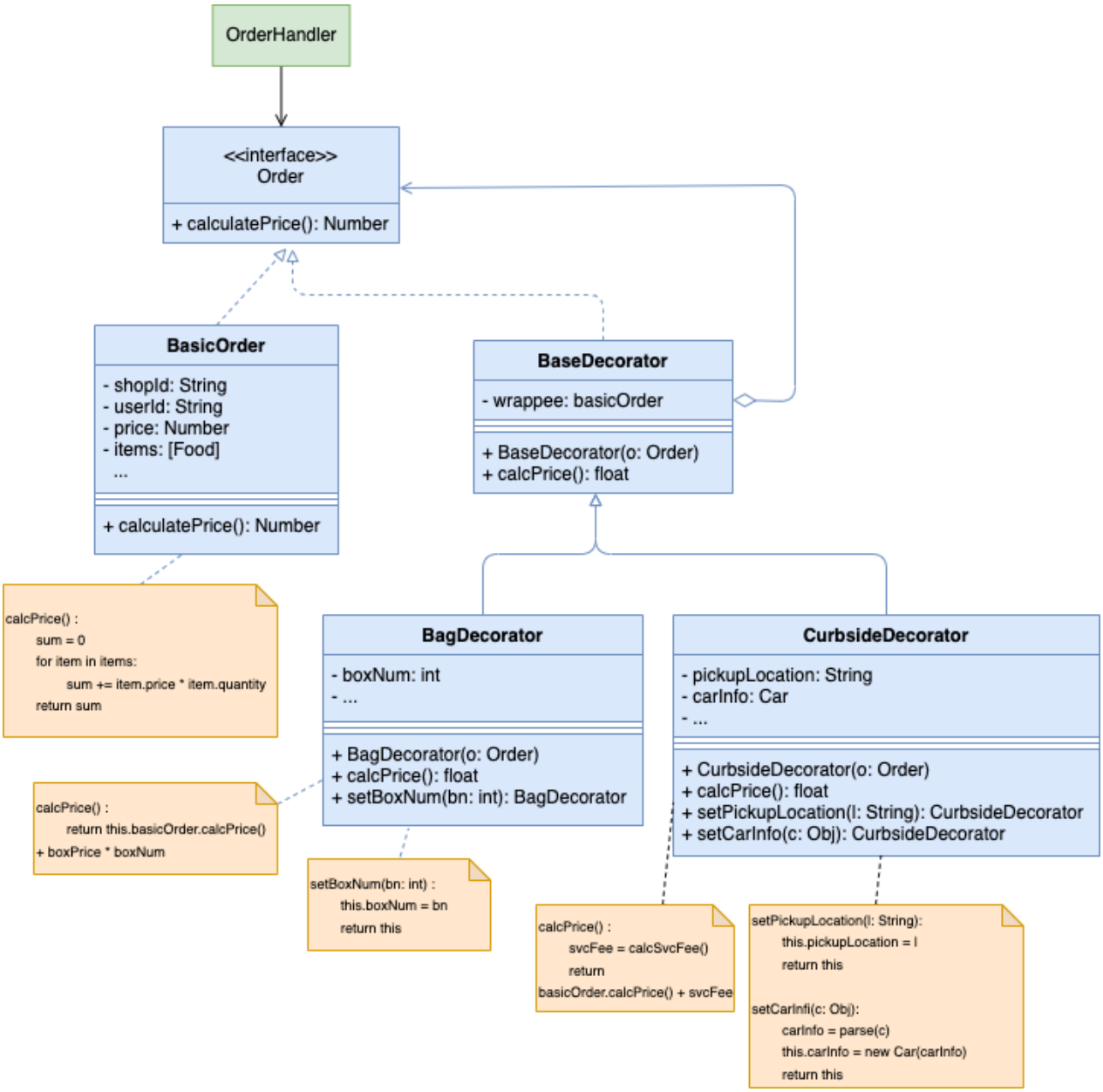

Figure 5.10: Class diagram of Decorator pattern in SharePlate

Problem / Rational

With the growth of the business, we want to exploit new order types based on the existing basic one. In this project, we are going to develop togo-box orders and curbside pickup orders, and in the future, we can foresee that we will have delivery orders based on the previous types. For each of the new order types, we want to add new features based on existing ones without changing the underlying code.

Benefits

- Single Responsibility Principle: Each decorator class has its own responsibility and doesn't interfere with the responsibilities of other classes.
- Open-Closed Principle: Add additional functionality to an object without modifying its underlying code.
- Flexible and low coupling: Extend the function of objects more flexibly than inheritance

Implementation

First I write a class called **`BasicOrder`** with a method **`calculatePrice()`**, which functions as the basic order type to be injected into different decorators. Since JavaScript doesn't support interfaces at language level, the order interface is not implemented explicitly.

`BaseDecorator` is the ancestor class of all the decorators. It takes the basic as a parameter when initializing, and puts it in a wrappee. It has an abstract method **`calculatePrice()`** to be implemented by its child class. However, since JavaScript doesn't support abstract functions, it will throw an error with the message **`UNIMPLEMENTED`** if called without being overridden.

`BagDecorator` and **`CurbsideDecorator`** are 2 concrete decorators. They inherit from **`BaseDecorator`**, and have new methods supporting new features. They implement **`calculatePrice()`** and leverage the basic order as the foundation.

In **`OrderHandler`**, where the system creates specific orders, we extract certain fields from the meta-information in the request body and create a new basic order instance. Then it will update the corresponding decorator based on the **`type`** field in the request body and inject the basic order into the decorator.

Source Code

This pattern is implemented in `s23-t2-shareplate-backend/app/handlers/orderHandler`

- Order Handler
  - `orderHandler.js`
- `Order Types`
  - `orderTypes/basicOrder.js`
  - `orderTypes/baseDecorator.js`
  - `orderTypes/toGoBoxDecorator.js`
  - `orderTypes/curbsideDecorator/curbsideDecorator.js`

Additional Details

Section 6: Package View

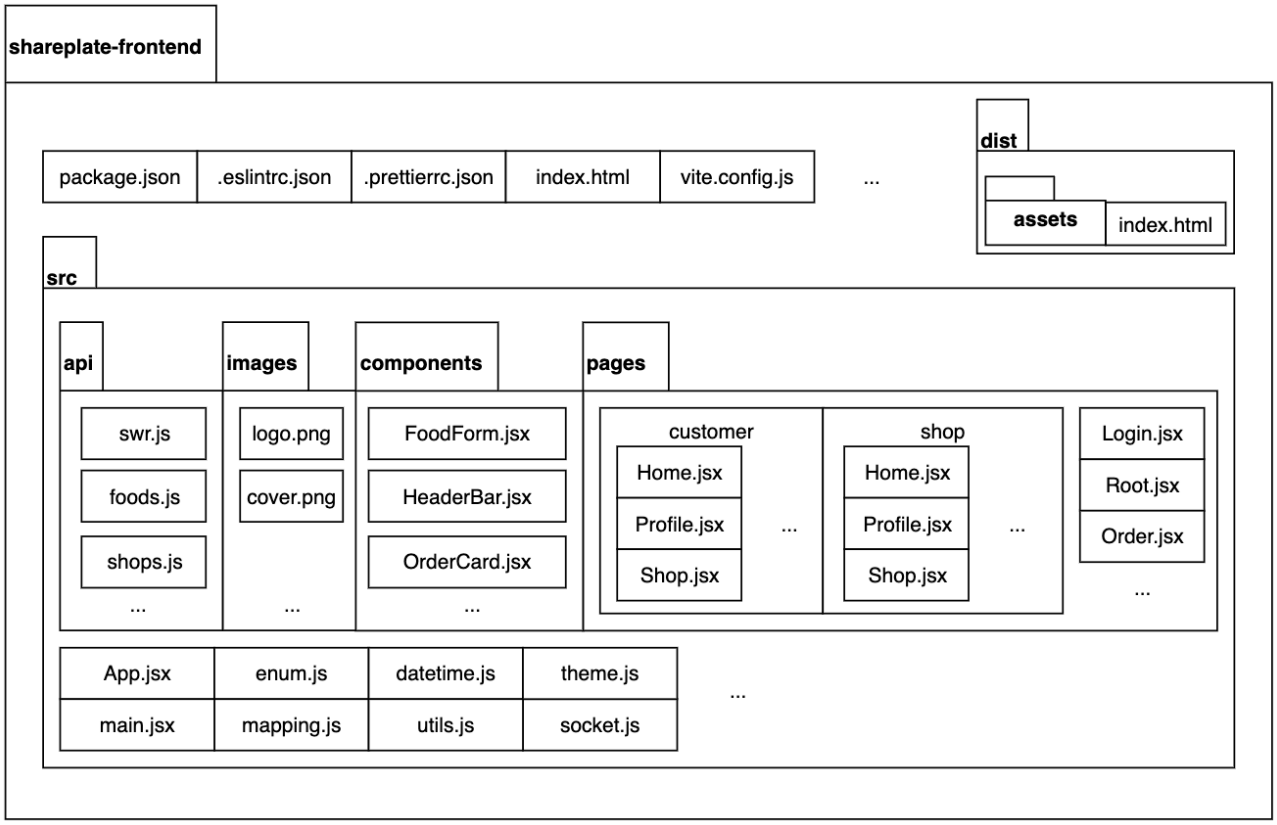

Figure 6.1: Frontend Package Diagram

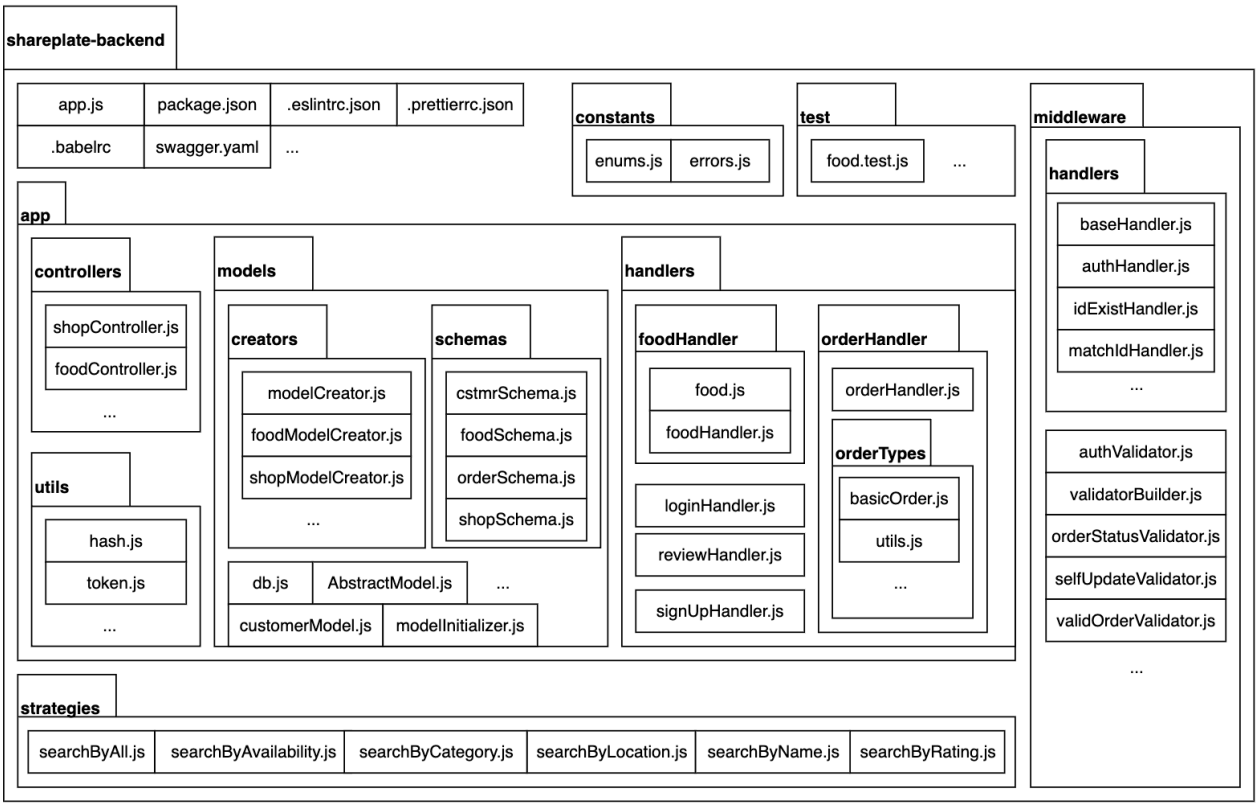

Figure 6.2: Backend Package Diagram

Section 7: Deployment View

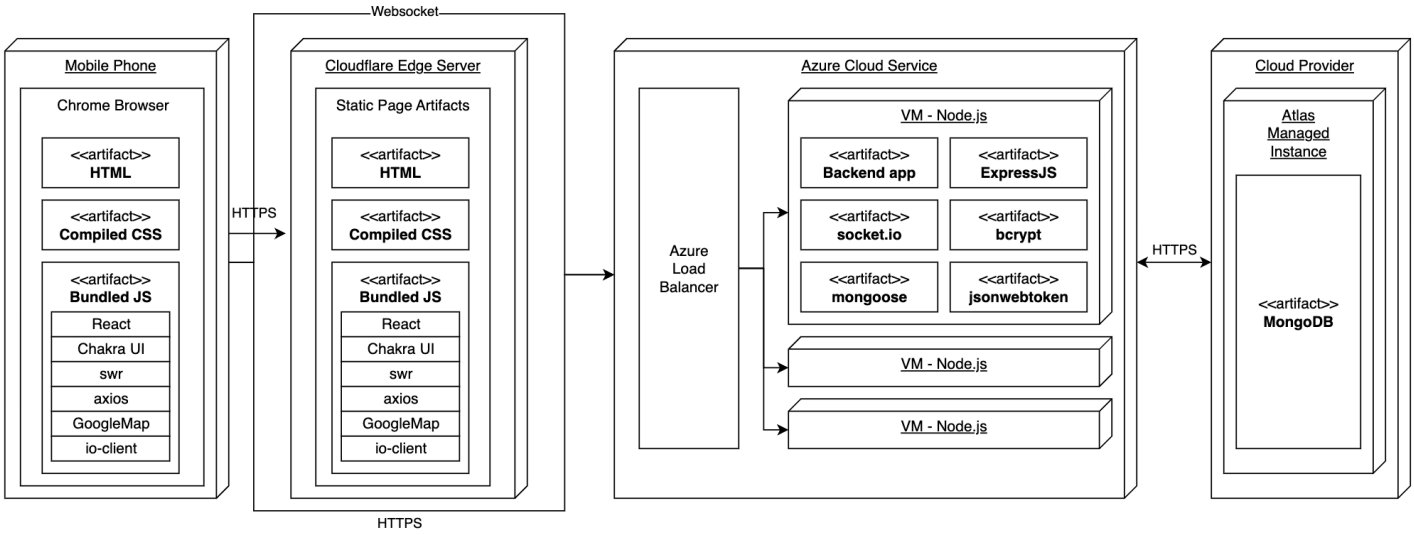

Figure 7.1: SharePlate Service Deployment View