SharePlate - Team 2

Responsibilities of main modules and components:

- **Front-End**
  - **SignUpView (Customer/Vendor)**: It is responsible for displaying the login and registration view. Also, it handles user login and registration events.
  - **LoginView (Customer/Vendor)**: It is responsible for displaying the login and registration view. Also, it handles user login and registration events.
  - **FoodManagementView (Vendor)**: It is responsible for displaying the food management interface. Vendors can post, edit or remove their food posting information here.
  - **FoodSearchView (Customer)**: It is responsible for showing the search result of foods in two different views: List view and Map view. Customers can use this module to view and select the food they want to reserve.
  - **OrderManagementView (Customer/Vendor)**: It is responsible for displaying past, present, and future orders. Customers can use this module to view order details and toggle the pickup code. Vendors can use this module to see all orders and scan the pickup code provided by the customer.
  - **FeedbackView (Customer/Vendor)**: It is responsible for displaying a rating user interface. Customers can use it to rate a vendor after they picked up the food. Vendors can use it to report no-show customers.

- **Back-End**
  - **Controllers**
    - **SignUpController**: This controller is in charge of signup. After a user registers, this component verifies that the username and password are still valid and stores the login information in the database.
    - **LoginController**: The login and logout processes will be handled by this controller. It will use the REST API to interact with the User Module.
    - **PostFoodController**: This module will handle the food post request from the front-end and communicate with FoodCollection in the database through interaction with FoodModel.
    - **SubscriptionController**: This controller will address and handle the subscription request from the user and establish bi-directional link records between the specific user with a certain shop.
    - **NotificationController**: This controller deals with the notification behaviors in the application, including receiving notifications requests from vendors or broadcasting notifications to users.
    - **ReviewController**: This controller will handle user review-related requests, including receiving review requests from users, formatting, and storing them in the database. Also, it can pull the reviews and send them back to the front-end via RESTful API.
    - **OrderController**: The controller will receive and handle the order request, processing and interacting the requests with order information in the database through OrderModel.
    - **SearchController:** This controller will receive user searching requests and interact with foodModel, retrieving specific food posts and sending the query results back to the front-end via RESTful API.
  - **Model**
    - **UserModel**: This model is responsible for communicating with the stored user collection in the database.
    - **ShopModel**: The model handles communicating with the shop collection in the database, executing behavior such as linking users to certain shop collections or deleting certain users in the specific shop collection.
    - **FoodModel**: The model handles the read, and write requests for food posts, interacting with the back-end food collection.
    - **OrderModel**: The model handles read and write requests from OrderComponent to retrieve order data from the database.

Technology stack

- **Frontend - Javascript + React:** We decided to use JavaScript and a popular JavaScript library React to create a dynamic and interactive application. React allows us to create reusable components, which improves reusability, helps speed up development time, and makes our code more maintainable.
- **Backend - Node.js + Express:** To enhance our compatibility with frontend technologies, we chose to use a JavaScript runtime environment Node.js. Express is a popular web application framework built on top of Node.js that simplifies the process of creating APIs and handling HTTP requests and responses, and we selected it to facilitate our backend development.
- **Database - MongoDB:** In a continuously developing process, a NoSQL database gives our application a higher scalability. MongoDB, in particular, can be easily connected to Node.js through the use of a third-party library Mongoose. In addition, as a cloud-based database service, MongoDB Atlas ensures high availability for our application.
- **Testing - Jest:** In order to achieve high testing efficiency, we chose to use Jest, a JavaScript testing framework that provides a simple and intuitive way to write tests for JavaScript code. Jest also provides great convenience in testing React components.
- **CI/CD - GitHub Action:** GitHub Actions is a powerful automation tool that allows us to set up continuous integration and continuous deployment pipelines directly from our GitHub repository, and it automates our building, deploying, and testing process.

Quality attributes

- **Availability**: SharePlate is a mission-critical application that must be highly available to meet the demands of its users. The target customers of SharePlate are food providers and customers looking for affordable and fresh food options. In order to provide a seamless experience to our customers, it is important that the application is always available. To achieve this, we made the following tech decisions:
  - We use AWS to host our application, leveraging a load balancer to distribute incoming traffic across multiple instances, which provides exceptional levels of high availability.
  - We use MongoDB Atlas, a cloud-based database service to ensure that our data is always available and replicated across multiple servers in different regions. MongoDB claims that Atlas clusters are highly available and backed by an industry-leading uptime SLA of 99.995% across all cloud providers.

- **Reusability**: SharePlate's code is designed to be modular and reusable, which helps to reduce development time and improve code quality. This is important to make sure that new features can be added quickly and that the codebase can be maintained in the long run. We made the following technical decisions to achieve this:
  - We use React as our front-end framework, which enables the development of reusable and modular components. This makes it easy to build and maintain a scalable and flexible front-end architecture.
  - We use middleware to enable the creation of reusable backend components. This helps to improve the modularity and maintainability of our backend code.

- **Testability**: SharePlate's system is easy to test, which ensures the quality and reliability of the platform. It is critical that the application is tested and reliable to ensure that customers have a good experience. To achieve this, we have made the following tech decisions:
  - We use dependency injection techniques to reduce coupling between components and make our code more testable. By using dependency injection, we can easily replace a real dependency with a mock object. This allows us to isolate each component and test it independently, without relying on external dependencies.
  - We use mocking libraries, such as mockingoose, to create mock objects that simulate the behavior of real dependencies. This enables us to test each component in isolation, without affecting other components. For example, when testing the OrderController, we can create a mock FoodDAOComponent that simulates the behavior of the actual component. This allows us to test the behavior of the OrderController without relying on the actual FoodDAOComponent.

Major design decisions with rationale

- **MVC architecture:** The MVC architectural design was selected for three key reasons:
  - Team development capability: The front-end and back-end development capabilities are clearly separated in our team's areas of specialty, therefore the MVC design is a good fit based on team capability.
  - Cost factors: For the scope of an application like SharePlate, which is relatively straightforward, using a Microservice or SOA design may be too complex and expensive for maintenance. MVC architecture, in comparison, is a better solution for a web application of this scope.
  - Application requirement: SharePlate should be able to maintain a short response time to handle a high volume of requests during mealtime. We don't need to over-define many interfaces for our application, which might increase unnecessary overhead. In this context, the MVC architecture is a better solution to prevent unnecessary overhead while maintaining response performance.

  Model: We can specify different data models to handle the data access behaviors of different schemas in SharePlate, such as food posts, users, orders, and reviews.

  View: With the MVC structure, we can focus on making customized and reusable views for different user behaviors. For example, we will be able to make customized interfaces for food providers and customers.

  Controller: The MVC gives us a way to handle business logic by making controllers according to different use cases and handling customer input and interactions with models and views. For example, an orderController could be made to handle customer order use cases. A subscriptionController could be made to handle user subscription events.

- **MongoDB Atlas:** As we anticipate food providers would be able to post food in flexible ways, such as various product combos or different kinds of food properties (i.e. hot/cold beverages), we choose to employ a document-based database to adapt to various food posting demands from suppliers without often changing the schema. MongoDB is a great and reliable solution for us to adopt our database. In addition, MongoDB Atlas is a cloud-optimized solution for cloud deployment, which will be advantageous for the cloud deployment of sharePlate.

- **Google Maps API:** The use of the Google Maps API in the SharePlate service provides accurate, efficient, and customizable mapping and location data that can help to enhance the user experience and provide valuable context for the platform.

- **Protocol:**
  - **RESTful API with HTTP:** Provides a flexible and scalable way of accessing and exchanging data.
  - **WebSockets:** Allow real-time updates for all users. For example, customers will be able to fetch real-time notifications and relevant posted data.

Components diagram *(Please open this file in draw.io)*

https://drive.google.com/file/d/1Q7DPGnI5-KTiyj1ThtIJgYNwoVBR_P3S/view?usp=share_link