Low Level Design Doc

Ravindranath Ramanujam Loganathan
Poorna Priyan Boopathy
Deepak Kumar Y
TECH WIZARDS CMPE-202 Sec 02 - SW Systems Engr

1. Backend (Java Spring Boot)

Controller Layer: Handles HTTP requests

RestaurantController

- **Endpoints**:
    - GET /restaurants/search: Search restaurants by filters (name, cuisine, price, ratings, etc.).
    - GET /restaurants/{id}: Get restaurant details by ID.
    - POST /restaurants: Add a new restaurant (BusinessOwner only).
    - PUT /restaurants/{id}: Update restaurant details (BusinessOwner only).
    - DELETE /restaurants/{id}: Delete a restaurant (Admin/BusinessOwner).
    - GET /restaurants/owner/{ownerId}: Fetch all restaurants owned by a BusinessOwner.

ReviewController

- **Endpoints**:
    - GET /restaurants/{id}/reviews: Get all reviews for a specific restaurant.
    - POST /restaurants/{id}/reviews: Submit a review for a restaurant.
    - DELETE /restaurants/{id}/reviews/{reviewId}: Delete a review (BusinessOwner/Admin).

UserController

- **Endpoints**:
    - POST /users/register: Register a new user (RegularUser or BusinessOwner).
    - POST /users/login: Login and generate JWT token.
    - GET /users/{id}: Get user details.
    - PUT /users/{id}: Update user details (Profile management).
    - DELETE /users/{id}: Delete a user (Admin only).
    - GET /users/{id}/reviews: Get all reviews submitted by a user.

AdminController

- **Endpoints**:

- GET /admin/duplicates: Fetch potential duplicate restaurants.
- DELETE /admin/restaurants/{id}: Remove restaurant (Admin).
- GET /admin/users: Fetch all users with roles (for management).
- DELETE /admin/users/{id}: Delete user (Admin).

Service Layer: Handles business logic

RestaurantService

- **Methods**:
  - searchRestaurants(filters): Searches for restaurants based on filters.
  - getRestaurantById(id): Retrieves restaurant details.
  - addRestaurant(restaurant): Adds a new restaurant.
  - updateRestaurant(id, restaurant): Updates an existing restaurant.
  - deleteRestaurant(id): Deletes a restaurant.
  - findDuplicates(): Finds potential duplicate restaurants based on name/address similarity.
  - getRestaurantsByOwner(ownerId): Retrieves all restaurants owned by a specific BusinessOwner.

ReviewService

- **Methods**:
  - getReviewsForRestaurant(restaurantId): Retrieves all reviews for a restaurant.
  - submitReview(restaurantId, review): Submits a new review for a restaurant.
  - deleteReview(reviewId): Deletes a review (Admin or BusinessOwner).

UserService

- **Methods**:
  - registerUser(user): Registers a new user or business owner.

- authenticateUser(credentials): Authenticates a user and generates a JWT token.
- getUserById(id): Retrieves user details.
- deleteUser(id): Deletes a user.
- updateUser(user): Updates user profile details.

AdminService

- **Methods**:
  - getDuplicateRestaurants(): Fetches potential duplicates.
  - getAllUsers(): Retrieves all users for management.
  - deleteUser(id): Deletes a user.
  - deleteRestaurant(id): Deletes a restaurant.

Repository Layer: Interacts with the database (using Spring Data JPA)

RestaurantRepository

- **Methods**:
  - findByNameOrAddress(name, address): Searches for duplicate restaurants based on name/address similarity.
  - findByCuisineAndFilters(cuisine, filters): Retrieves restaurants by cuisine, price, rating, and other filters.
  - findByOwnerId(ownerId): Retrieves restaurants owned by a specific BusinessOwner.

ReviewRepository

- **Methods**:
  - findByRestaurantId(restaurantId): Fetches reviews for a specific restaurant.
  - deleteById(reviewId): Deletes a specific review.

UserRepository

- **Methods**:

- findByUsername(username): Retrieves user by username.
- findAll(): Retrieves all users (for admin).
- findById(id): Retrieves user by ID.

---

2. Frontend (ReactJS)

Pages

HomePage (Search Page)

- **Components**:
    - SearchBar: Allows users to search restaurants by name, cuisine, price, etc.
    - Filters: Dropdowns for cuisine, price, rating.
    - RestaurantList: Displays a list of restaurants based on the search query.
- **API Calls**:
    - GET /restaurants/search: Triggered when a user searches for restaurants.

RestaurantDetailsPage

- **Components**:
    - RestaurantInfo: Displays restaurant name, address, contact info, and description.
    - PhotosCarousel: Carousel for restaurant photos.
    - Reviews: Lists all reviews for the restaurant.
    - SubmitReviewForm: Allows users to submit a review.
- **API Calls**:
    - GET /restaurants/{id}: Fetch restaurant details.
    - GET /restaurants/{id}/reviews: Fetch reviews for the restaurant.
    - POST /restaurants/{id}/reviews: Submit a new review.

LoginPage

- **Components**:
    - LoginForm: Handles username/password input and authentication.
- **API Calls**:
    - POST /users/login: Authenticate user and get JWT token.

RegistrationPage

- **Components**:

    - UserRegistrationForm: Handles new user registration.
    - BusinessOwnerForm: Conditionally displayed if the "Business Owner" checkbox is selected.
- **API Calls**:
    - POST /users/register: Register new user or business owner.

AdminDashboardPage

- **Components**:
    - DuplicateListings: Displays potential duplicate listings.
    - AllRestaurants: Displays all restaurants with delete options.
    - UserManagement: Displays user list with management options.
- **API Calls**:
    - GET /admin/duplicates: Fetch duplicate listings.
    - GET /admin/restaurants: Fetch all restaurant listings.
    - GET /admin/users: Fetch all users.

UserManagementPage (Admin)

- **Components**:
    - UserList: Displays users with their roles (Regular User, Business Owner) and delete options.
- **API Calls**:
    - DELETE /admin/users/{id}: Delete a user.

---

3. Database Design (MySQL)

Tables

1. **users**
    - id (Primary Key)
    - username (Unique)
    - email
    - password (Hashed)
    - role (ENUM: RegularUser, BusinessOwner, Admin)
    - created_at
2. **restaurants**
    - id (Primary Key)
    - name
    - address
    - contact_info
    - cuisine (ENUM: Italian, Chinese, etc.)
    - food_type (ENUM: Vegetarian, Vegan, Gluten-Free)
    - price_range (ENUM: $, $$, $$$, $$$$)
    - rating
    - business_owner_id (Foreign Key to users.id)
    - created_at
3. **reviews**
    - id (Primary Key)
    - restaurant_id (Foreign Key to restaurants.id)
    - user_id (Foreign Key to users.id)
    - rating (1-5 stars)
    - comment
    - created_at

4. **admin_logs**
    - id (Primary Key)
    - admin_id (Foreign Key to users.id)
    - action (ENUM: DeleteRestaurant, DeleteUser)
    - target_id (Can be restaurant ID or user ID)
    - created_at

---

4. Flow of Operations

1. User Registration

- User fills out the registration form.
- Frontend sends POST /users/register to the backend.
- Backend saves the new user or business owner to the **user's** table.
- Confirmation response is sent to the frontend.

2. Restaurant Search

- User searches for restaurants using filters.
- Frontend sends GET /restaurants/search with query parameters.
- Backend queries the **restaurants** table using filters and returns matching results.
- Frontend displays the results in a list.

3. Admin Duplicate Check

- Admin views potential duplicate listings.
- Frontend sends GET /admin/duplicates to fetch potential duplicates.
- Backend queries **restaurants** for similar names/addresses and returns the data.
- Admin can delete duplicates, which triggers a DELETE /admin/restaurants/{id} API call.

API Definitions (REST Endpoints)

- **GET /restaurants/search**: Accepts query parameters for filtering (e.g., cuisine, price, rating).
- **GET /restaurants/{id}**: Returns detailed information about a restaurant.
- **POST /restaurants**: Business owners can add new restaurants.
- **PUT /restaurants/{id}**: Updates restaurant information.
- **DELETE /restaurants/{id}**: Admins or BusinessOwners can remove restaurants.
- **GET /restaurants/{id}/reviews**: Fetches reviews for a restaurant.
- **POST /restaurants/{id}/reviews**: Allows users to submit reviews.
- **POST /users/register**: Registers users and business owners.

- **POST /users/login**: Authenticates users and returns a JWT token.
- **GET /admin/duplicates**: Admins can check for duplicate listings.
- **GET /admin/users**: Admins can fetch all user data.
- **DELETE /admin/users/{id}**: Admins can delete a user.
- **GET /restaurants/owner/{ownerId}**: Business owners can fetch their own restaurant listings.

Security Layer (Authentication and Authorization)

- **JWT (JSON Web Tokens)**: Used for authentication and authorization. After successful login, the user receives a JWT token.
- **Spring Security**: Configured to protect API endpoints based on user roles (Admin, BusinessOwner, RegularUser). (Optional if we had further time for development)

Error Handling

Effective error handling ensures that the system gracefully responds to issues, providing meaningful feedback to both users and developers.

Backend Error Handling

- **Global Exception Handling**: Use Spring Boot's @ControllerAdvice and @ExceptionHandler annotations to create a global exception handler.
    - Example Errors:
        - **400 (Bad Request)**: For invalid user input (e.g., missing required fields, invalid data formats).
        - **401 (Unauthorized)**: For invalid authentication attempts (e.g., invalid JWT token).
        - **403 (Forbidden)**: When users try to access resources they don't have permissions for (e.g., a regular user trying to delete a restaurant).
        - **404 (Not Found)**: When a requested resource (e.g., a restaurant, review, or user) does not exist.

        - **500 (Internal Server Error)**: Catch-all for unhandled exceptions that result in server-side errors.

- **Validation Errors**: Use @Valid and @NotNull annotations in the request objects (DTOs) to validate user input, and provide clear validation messages when data doesn't meet the criteria.

Frontend Error Handling

- Use **React's Error Boundaries** to catch JavaScript errors and display a fallback UI when components fail to render.
- Display meaningful error messages when API requests fail (e.g., show a "Restaurant not found" message when a 404 error occurs).

5. Map Box API Integration for Maps

Using Map API for Fetching the Information:

Requirement Overview:

- **Search by ZIP Code**: Users should be able to search for restaurants by ZIP code.
- **Fetch Restaurants**: The search should return restaurants listed in the database as well as those fetched from external sources (Mapbox) that are not listed by the BusinessOwner in the application.
- **Maps API**: Integrate with Mapbox API to display search results.

1. Backend (Java Spring Boot)

Controller Layer

GeocodingController

- **Endpoints**:
    - GET /geocode/{zipcode}: Fetch latitude and longitude coordinates for a given ZIP code using the Mapbox Geocoding API.
    - **Logic**:
        - This endpoint will first call Mapbox's Geocoding API to get the coordinates of the ZIP code.
        - It will then use those coordinates to search for restaurants both in the local database (from BusinessOwners) and externally using the Mapbox Places API.

RestaurantSearchController

- **Endpoints**:
    - GET /restaurants/zipcode/{zipcode}: Search restaurants by ZIP code.
    - **Logic**:
        - First, fetch restaurants in the database that match the ZIP code.
        - Then, fetch additional restaurant information from Mapbox Places API for external restaurants not listed by BusinessOwners.

Service Layer

MapboxGeocodingService

- **Methods**:
    - getCoordinates(zipcode): Calls the Mapbox Geocoding API to convert a ZIP code into latitude and longitude.
    - getNearbyRestaurants(lat, lon): Calls the Mapbox Places API to search for restaurants near the provided latitude and longitude.

RestaurantSearchService

- **Methods**:
    - searchRestaurantsByZip(zipcode): First searches the local database for restaurants, then uses MapboxGeocodingService.getNearbyRestaurants to fetch external restaurants.

External Mapbox API Calls

Geocoding API

- **Purpose**: Convert the ZIP code entered by the user into geographic coordinates.
- Endpoint:

GET
https://api.mapbox.com/geocoding/v5/mapbox.places/{zipcode}.json?access_token={YOUR_MAPBOX_ACCESS_TOKEN}

- **Response**: Returns the latitude and longitude for the provided ZIP code.

Mapbox Places API

- **Purpose**: Search for nearby restaurants using coordinates.
- **Endpoint**

GET
https://api.mapbox.com/geocoding/v5/mapbox.places/restaurant.json?proximity={longitude},{latitude}&access_token={YOUR_MAPBOX_ACCESS_TOKEN}

- **Response**: Returns nearby restaurants from external sources.

2. Frontend (ReactJS)

Search Results Page

- **Components**:
    - SearchBar: Allows users to search for restaurants by ZIP code.
    - MapboxMap: Displays the search results on a map.
    - RestaurantList: Displays restaurant names and addresses from both internal (BusinessOwner listed) and external (Mapbox Places API) sources.

Flow of Data:

1. **User Search**:
    - User enters a ZIP code into the SearchBar.
2. **API Call**:
    - The frontend sends a request to /restaurants/zipcode/{zipcode}.
3. **Backend Process**:
    - The backend first checks its database for restaurants.
    - It then calls the Mapbox Places API to fetch nearby restaurants not listed by BusinessOwners.
4. **Display Results**:
    - The frontend renders both internal and external restaurants on the map using the **MapboxMap** component, with markers for each restaurant.

3. Flow of Operations for map

1. Search by ZIP Code

1. **User Action**: The user enters a ZIP code and submits the search.

2. **Frontend**:
    - Sends a GET /restaurants/zipcode/{zipcode} request to the backend.
3. **Backend**:
    - **Step 1**: Calls Mapbox Geocoding API to get latitude and longitude for the ZIP code.
    - **Step 2**: Queries the database for restaurants listed by BusinessOwners in the specified ZIP code.
    - **Step 3**: Calls the Mapbox Places API to get nearby restaurant data that are not listed in the database.
    - **Step 4**: Combines both results (internal database + external data) and sends the combined list back to the frontend.
4. **Frontend**:
    - Displays both internal (BusinessOwner listed) and external (Mapbox Places API) restaurants on the map and in a list view.

4. Database Design Update

To support the ZIP code search for restaurants listed by BusinessOwners, make sure your **restaurants** table includes a **zipcode** column:

restaurants Table (MySQL)

- **Fields**:
    - id: Primary Key
    - name: Restaurant name
    - address: Restaurant address
    - zipcode: ZIP code
    - cuisine: Cuisine type (e.g., Italian, Chinese)
    - price_range: Price level (Low, Medium, High)
    - business_owner_id: Foreign Key to users table (optional, for BusinessOwner listed restaurants)

----------END----------