Design Document

Food Order App Backend Server

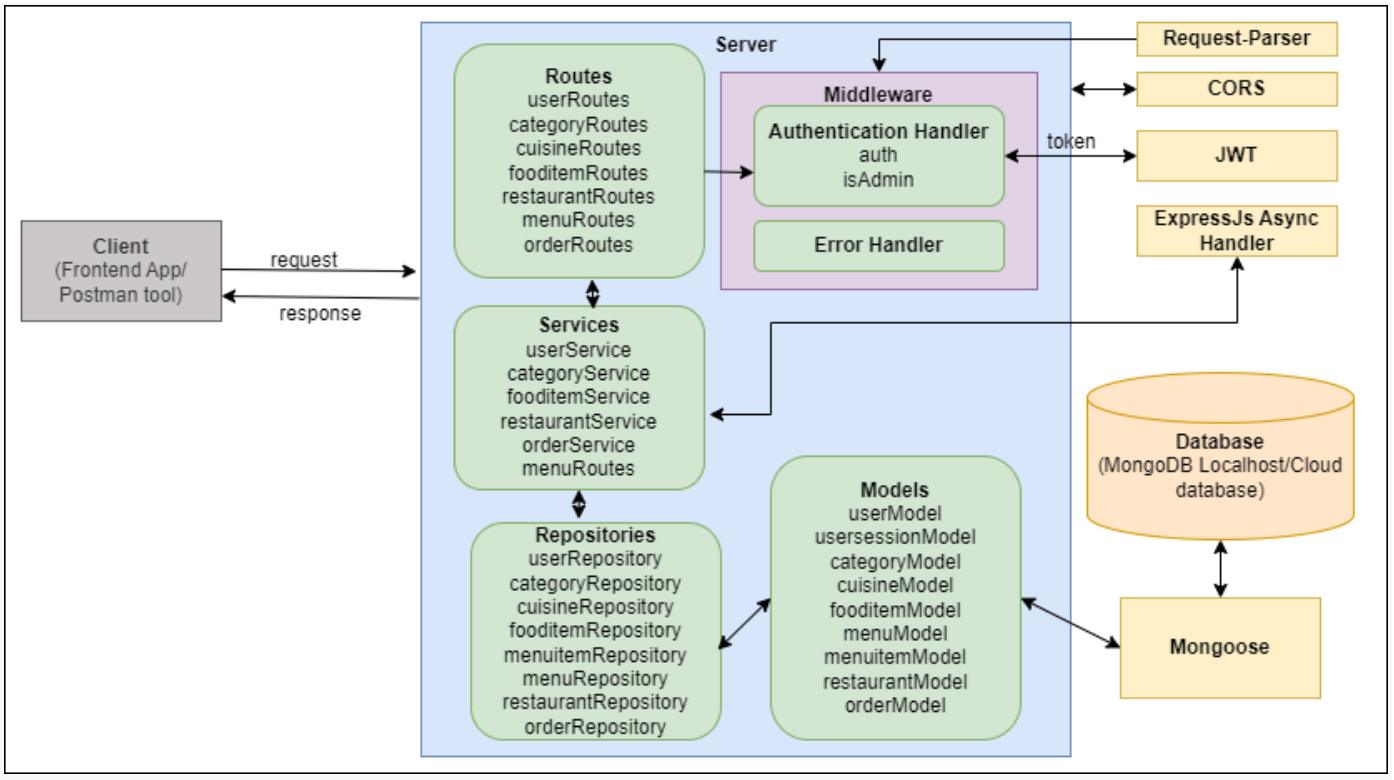
Table of Content

Introduction
Architecture of FoodOrderApp Backend Server
Setup and Configuration
    Prerequisites
    Installation
User Flow and Access Control
Sample Flow
API Endpoints
    User Routes
    Category Routes
    Cuisine Routes
    Fooditem Routes
    Restaurant Routes
    Menu Routes
    Order Routes
Data Models
    User
    UserSession
    Category
    Cuisine
    Fooditem
    Menu
    Menuitem
    Order
    Restaurant
Security and Authentication
Error Handling
Conclusion

Introduction

This document outlines the design and architecture of the backend for a foodorder application built with ExpressJS. It details the setup, routes, data models, and security measures implemented to support the application's functionality.

Architecture of FoodOrderApp Backend Server



The server application has various components including middleware for authentication and error handling, and a data handling flow from routes to database as follows :

- **Client**: Represents the external entity that makes requests to and receives responses from the server application.
- **Server Application**: The main block that encapsulates all server-side logic.
    - **Middleware**:
        - **Authentication Handler**: Manages authentication, verifying requests based on JWT tokens.
        - **Error Handler**: Manages error handling across the server application.
    - **Routes**: Defines various endpoints the server exposes to the client.
    - **Services**: Contains business logic that processes data, received from routes, before it's passed to repositories or vice versa.
    - **Repositories**: Acts as a data access layer, where the actual database CRUD (Create, Read, Update, Delete) operations are implemented.

- **Models**: Represents the structure of the data in the database; this is where you define your data schemas.

- **JWT**: An external library that generates and validates JSON Web Tokens for authentication purposes.
- **CORS**: An external library that handles Cross-Origin Resource Sharing to allow secure communication between client and server.
- **Express-Async-Handler**: An external library that streamlines asynchronous operations and error handling within Express.js route handlers, enhancing code structure and readability.
- **Mongoose:** Facilitates data modeling and interaction with the MongoDB database, enhancing the backend architecture's efficiency and robustness.
- **Database**: The MongoDB database where data is stored. It can be hosted locally or in the cloud.
- **Request-Parser**: It is a piece of middleware in a Node.js application that intercepts incoming HTTP requests, parses the request body (if any), and prepares the request object with the parsed data before passing it to the next middleware or route handler.

This architecture supports a clean separation of concerns, with each component focusing on a specific responsibility. The flow of data from routes to the database (and back) is a common pattern in web applications, facilitating maintainability and scalability.

Setup and Configuration

Prerequisites

Node.js installed
MongoDB for the database
Environment variables for database URI, secret keys, etc.

Installation

1. Clone the repository
2. Run **npm install** to install dependencies
3. Start the server with **npm start**
4. Access the server at **http://localhost:8080**

User Flow and Access Control

1. Create a user account (admin or non-admin).
2. Log in with the created user credentials.
3. Admin users have access to CRUD operations on users, categories, cuisines, foodItems, menuItems and orders.
4. Non-admin users have restricted access and can fetch the categories, cuisines, foodItems, restaurants and menus. They can also place orders and see their order details.
5. Once logged out, only operations that do not require authentication can be performed.

Sample Flow

1. Create a user using the HTTP POST request http://localhost:8080/api/v1/users with http body -

```
{
    "username":"john_d",
    "fullname" : "John Doe",
    "email" : "johnd@gmail.com",
    "isAdmin" : true,
    "password" : "johnd"
}
```

2. On successful creation of user, now login with user credentials using HTTP POST request http://localhost:8080/api/v1/users/login with http body -

```
{
    "username":"john_d",
    "password" : "johnd"
}
```

3. On successful login, copy the session token generated and paste it in the Authorization tab (Authorization Type should be Bearer Token) in the Token input field.
4. Now since you have successfully logged in the user credentials who is Admin so you should be able to get all users list or generate a new category, a new cuisine or new fooditem etc.
5. Refer to the endpoints mentioned below to perform all such operations and then you can give a logout request.

API Endpoints

User Routes

- **Register**: **POST /api/v1/users** - Creates a new user account.
- **Login**: **POST /api/v1/users/login** - Authenticates a user and returns a token.
- **Logout**: **PUT /api/v1/users/logout** - Logs out a user, requiring token authentication.
- **Fetch User**: **GET /api/v1/users/:id** - Retrieves user details, requires token authentication.
- **Delete User**: **DELETE /api/v1/users/:id** - Removes a user account, requires token authentication and admin rights.
- **Edit User**: **PUT /api/v1/users/:id** - Updates user details, requires token authentication.
- **Fetch All Users**: **GET /api/v1/users** - Retrieves all users, requires token authentication and admin rights.

Category Routes

- **Create Category**: **POST /api/v1/categories** - Adds a new category, requires token authentication and admin rights.
- **Edit Category**: **PUT /api/v1/categories/:id** - Updates category details, requires token authentication and admin rights.
- **Delete Category**: **DELETE /api/v1/categories/:id** - Removes a category, requires token authentication and admin rights.
- **Fetch Category**: **GET /api/v1/categories/:id** - Retrieves a category detail.
- **Fetch All Categories**: **GET /api/v1/categories** - Lists all categories.

Cuisine Routes

- **Create Cuisine**: **POST /api/v1/cuisines** - Adds a new cuisine, requires token authentication and admin rights.
- **Edit Cuisine**: **PUT /api/v1/cuisines/:id** - Updates cuisine details, requires token authentication and admin rights.
- **Delete Cuisine**: **DELETE /api/v1/cuisines/:id** - Removes a cuisine, requires token authentication and admin rights.
- **Fetch Cuisine**: **GET /api/v1/cuisines/:id** - Retrieves a cuisine detail.
- **Fetch All Cuisines**: **GET /api/v1/cuisines** - Lists all cuisines.

Fooditem Routes

- **Create Fooditem**: **POST /api/v1/fooditems** - Adds a new fooditem, requires token authentication and admin rights.
- **Edit Fooditem**: **PUT /api/v1/fooditems/:id** - Updates fooditem details, requires token authentication and admin rights.
- **Delete Fooditem**: **DELETE /api/v1/fooditems/:id** - Removes a fooditem, requires token authentication and admin rights.
- **Fetch Fooditem**: **GET /api/v1/fooditems/:id** - Retrieves a fooditem detail.
- **Fetch All Fooditems**: **GET /api/v1/fooditems** - Lists all fooditems.

Restaurant Routes

- **Create Restaurant**: **POST /api/v1/restaurants** - Adds a new restaurant, requires token authentication and admin rights.
- **Edit Restaurant**: **PUT /api/v1/restaurants/:id** - Updates restaurant details, requires token authentication and admin rights.
- **Delete Restaurant**: **DELETE /api/v1/restaurants/:id** - Removes a restaurant, requires token authentication and admin rights.
- **Fetch Restaurant**: **GET /api/v1/restaurants/:id** - Retrieves a restaurant detail.
- **Fetch All Restaurants**: **GET /api/v1/restaurants** - Lists all restaurants.
- **Fetch All Restaurants by Cuisine Name**: **GET /api/v1/restaurants?cuisine=<CuisineName> -** Lists all restaurants based on a cuisine.
- **Fetch All Restaurants by Category Name**: **GET /api/v1/restaurants?category=<CategoryName> -** Lists all restaurants based on a category.

Menu Routes

- **Create Menu**: **POST /api/v1/menus –** Adds a new menu, requires token authentication and admin rights.
- **Edit Menu**: **PUT /api/v1/menus/:id –** Updates menu details, requires token authentication and admin rights.
- **Delete Menu**: **DELETE /api/v1/menus/:id –** Removes a menu, requires token authentication and admin rights.

- **Fetch Menu Based on RestaurantId**: **GET /api/v1/menus/:restaurantId –** Retrieves a menu detail based on a restaurantId.

Order Routes

- **Place Order**: **POST /api/v1/orders** - Submits a new order, requires token authentication.
- **Get Order Details**: **GET /api/v1/:userId/:orderId –** Retrieves the order details of a user's order, requires token authentication.
- **User Orders**: **GET /api/v1/orders/:userId** - Retrieves a user's order history, requires token authentication.
- **All Orders**: **GET /api/v1/orders** - Lists order history of all users, requires token authentication and admin rights.
- **Change Order Status**: **PUT /api/v1/orders/:orderId** - Modifies the status of an order, requires token authentication and admin rights.

Data Models

User

- ID
- Username
- Fullname
- Email
- Password (hashed)
- Admin (boolean)

UserSession

- ID
- UserID
- SessionToken

Category

- ID
- Name
- Description
- Image

Cuisine

- ID
- Name
- Description
- Image

Fooditem

- ID
- Name
- Description
- Image
- CategoryID
- CuisineID
- IsVeg

Menu

- ID
- RestaurantID
- Description

Menuitem

- ID
- MenuID
- FooditemID
- FooditemName
- FooditemImage
- FooditemPrice

Order

- ID
- UserID
- RestaurantID
- Status (Processing, Completed, Cancelled)
- OrderTotalPrice
- Fooditems
- ShippingDetails

Restaurant

- ID
- Name
- Address
- Contact
- Image

Security and Authentication

- Implement JWT for user authentication and authorization.
- Admin routes require token verification and an admin flag check.

Error Handling

- Implement global error handling middleware for catching and responding to errors.

Conclusion

This design document outlines the key components of the backend for an ExpressJS-based Food Order application, including the setup, API endpoints, data models, and security considerations. This structure aims to provide a scalable, secure, and efficient backend system for managing foodOrder operations.