1. Comprehensive Website Handbook

1.1. Index

1.2. Introduction

1.2.1. About the Website

The website isa platformfor the userto order food online. The website provides a list of restaurants, their menus, and allows users to place orders for delivery or pickup or dine in . Users can create accounts, save their favorite orders, and track the status of their orders in real-time.

1.2.2. Purpose of the Handbook

The purpose of this handbook is to provide a comprehensive guide to the website's architecture, codebase, and functionality. It is intended for developers, testers, and other stakeholders who need to understand how the website works, how to set it up locally, and how to maintain and troubleshoot it.

1.2.3. Target Audience (Developers, Marketers, Testers, etc.)

The target audience for this handbook includes:

- Developers who need to understand the codebase, APIs, and database design.
- Testers who need to know how to test the website and report bugs.
- Marketers who need to understand the website's features and target audience.
- Project managers who need to oversee the development and deployment of the website.
- Non-technical staff who need a high-level overview of the website's functionality.
- New team members who need to onboard quickly and understand the project.
- Anyone interested in learning about web development and programming.

1.2.4. How to Use This Handbook

This handbook is organized into sections that cover different aspects of the website, from the high-level overview to the technical details of the codebase and database design. You can use the table of contents to navigate to specific sections or read through the entire handbook to get a comprehensive understanding of the website.

1.3. Website Overview

1.3.1. Vision and Mission of the Website

The vision of the website is to provide a seamless and convenient online ordering experience for users, connecting them with their favorite restaurants and enabling them to order food with ease. The mission of the website is to offer a wide variety of food options,

ensure timely delivery, and provide a user-friendly interface that makes ordering food a pleasant experience.

1.3.2. Key Features and Offerings

The website offers the following key features and offerings:

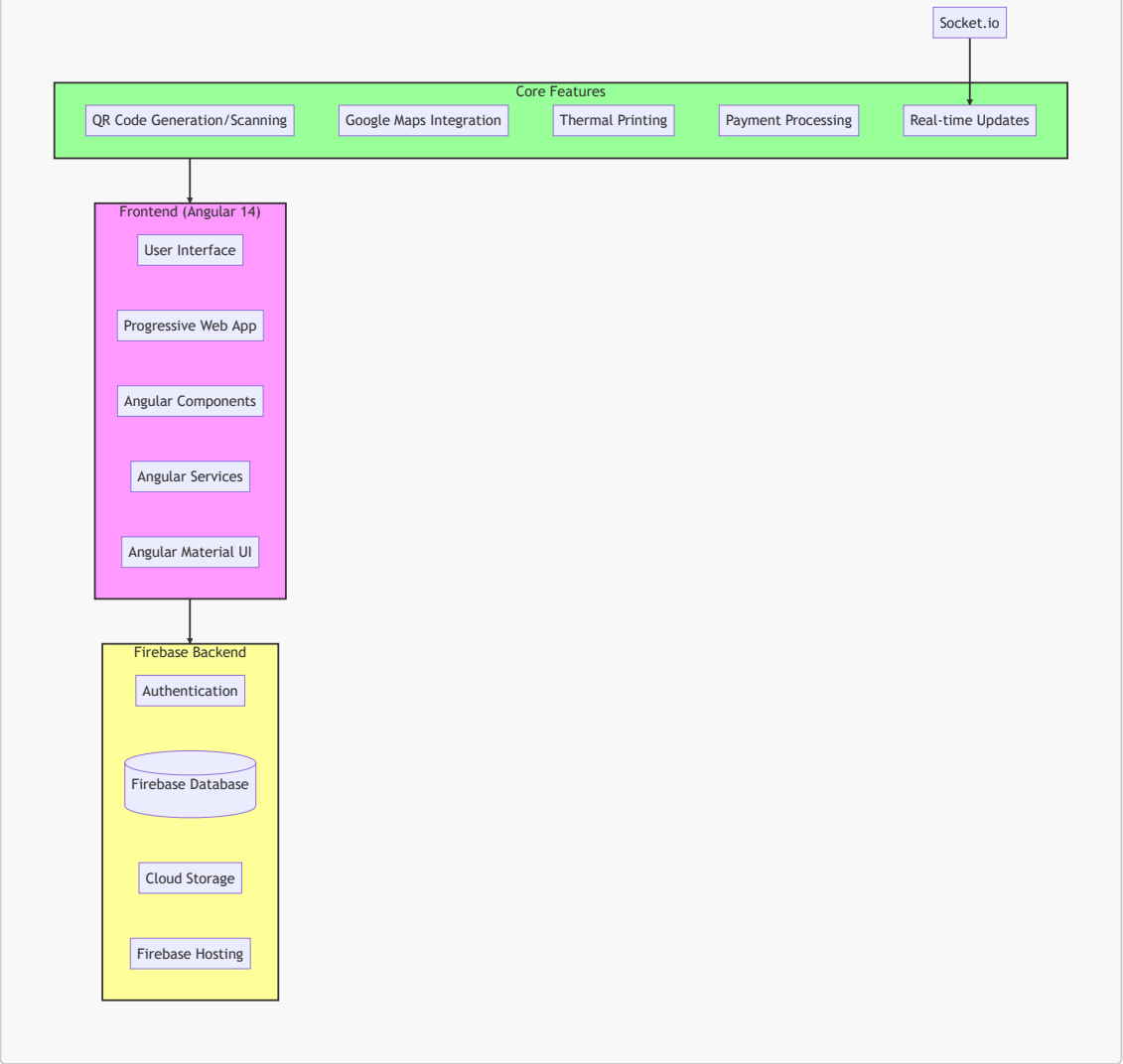
- User registration and account creation
- Restaurant listings with menus and reviews
- Order placement for delivery, pickup, or dine-in
- Real-time order tracking
- Favorite orders and reordering
- Payment gateway integration
- Messaging service integration for order updates

1.3.3. Target Audience for the Website (end-users)

The target audience for the website includes:

- Working professionals who want to order food for lunch or dinner
- Families looking to order meals for home delivery
- Students who want to order food for study sessions
- Tourists and travelers looking for local cuisine
- Food enthusiasts who want to explore new restaurants
- Event organizers who need catering services
- Anyone who prefers the convenience of online food ordering
- Anyone who wants to avoid the hassle of cooking

1.3.4. High-Level Overview of the Website Workflow



1.3.5. Glossary of Terms (for technical and non-technical users)

1.3.5.1. General Terms

- **Digital Menu**: An electronic version of a restaurant's menu that can be accessed through web browsers or mobile devices
- **POS (Point of Sale)**: The system where transactions are processed and orders are managed
- **QR Code**: A square barcode that can be scanned by smartphones to quickly access the digital menu
- **Cart**: A virtual collection of items selected by the customer before placing an order

1.3.5.2. Order Types

- **Dine-in**: Customers eating at the restaurant premises
- **Takeaway**: Customers picking up their order from the restaurant
- **Delivery**: Food being delivered to the customer's specified location

1.3.5.3. User Roles

- **Customer**: End-user who browses the menu and places orders
- **Restaurant Staff**: Personnel who manage orders and update menu items

- **Admin**: System administrator with full access to manage the platform
- **Delivery Partner**: Person responsible for delivering orders to customers

1.3.5.4. Technical Terms

- **Frontend**: The user interface that customers interact with (website/app)
- **Backend**: Server-side system that processes requests and manages data
- **API (Application Programming Interface)**: System that allows different parts of the application to communicate
- **Database**: System that stores all menu items, orders, and user information
- **Authentication**: Process of verifying user identity
- **Cache**: Temporary storage of frequently accessed data for faster performance

1.3.5.5. Payment Terms

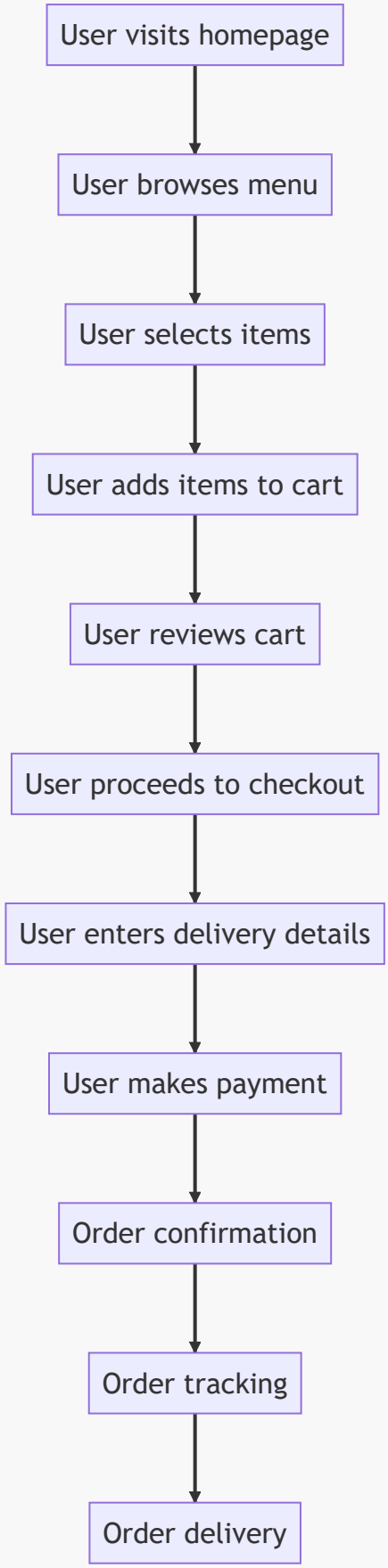
- **Payment Gateway**: System that processes online payments securely
- **Transaction**: A completed order payment
- **Payment Status**: Current state of payment (pending/completed/failed)
- **Refund**: Return of payment to customer's account

1.3.5.6. Features

- **Real-time Tracking**: Live monitoring of order status
- **Menu Customization**: Ability to modify menu items based on availability
- **Order History**: Record of all past orders
- **Favorites**: Saved list of frequently ordered items
- **Reviews & Ratings**: Customer feedback system

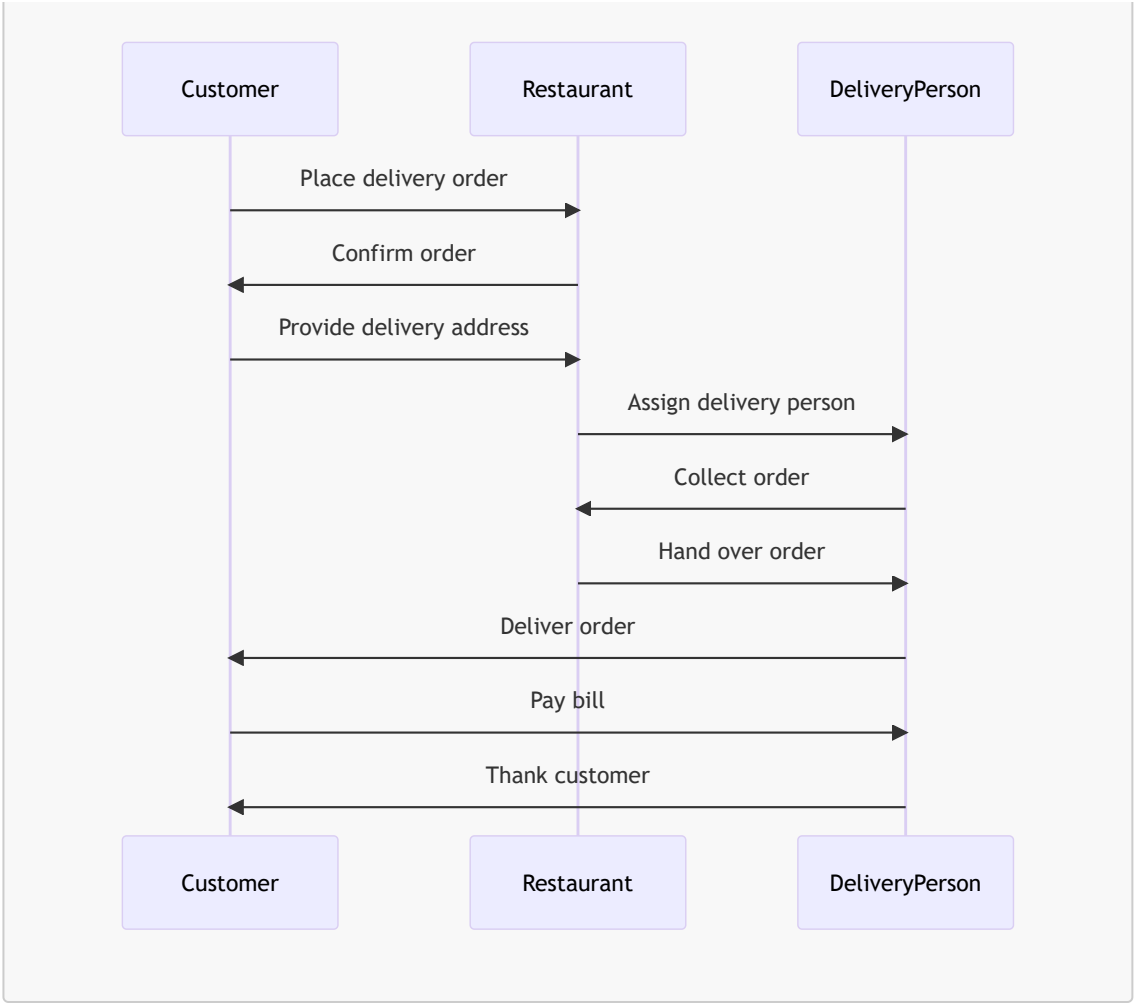
1.4. Functional Flow

1.4.1. User Flows

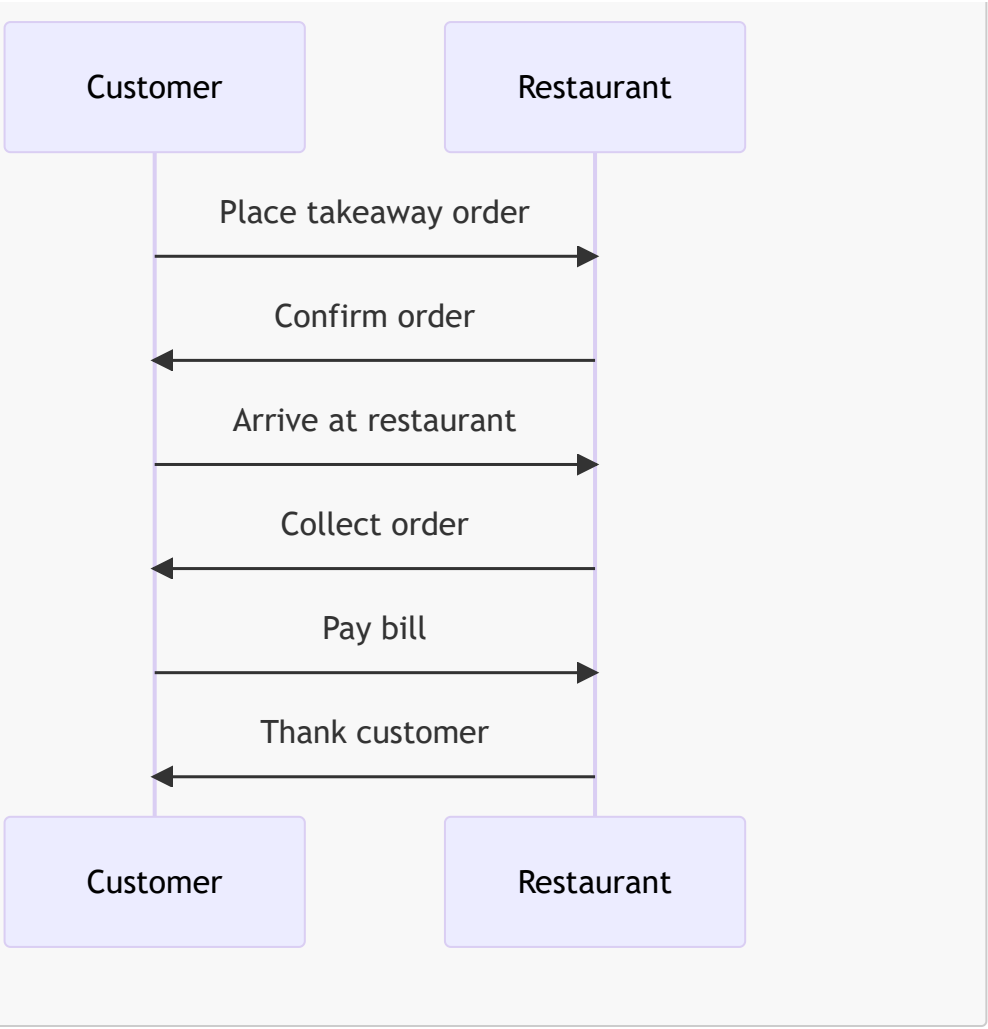


1.4.2. Visual Flow Diagrams for Each User Flow

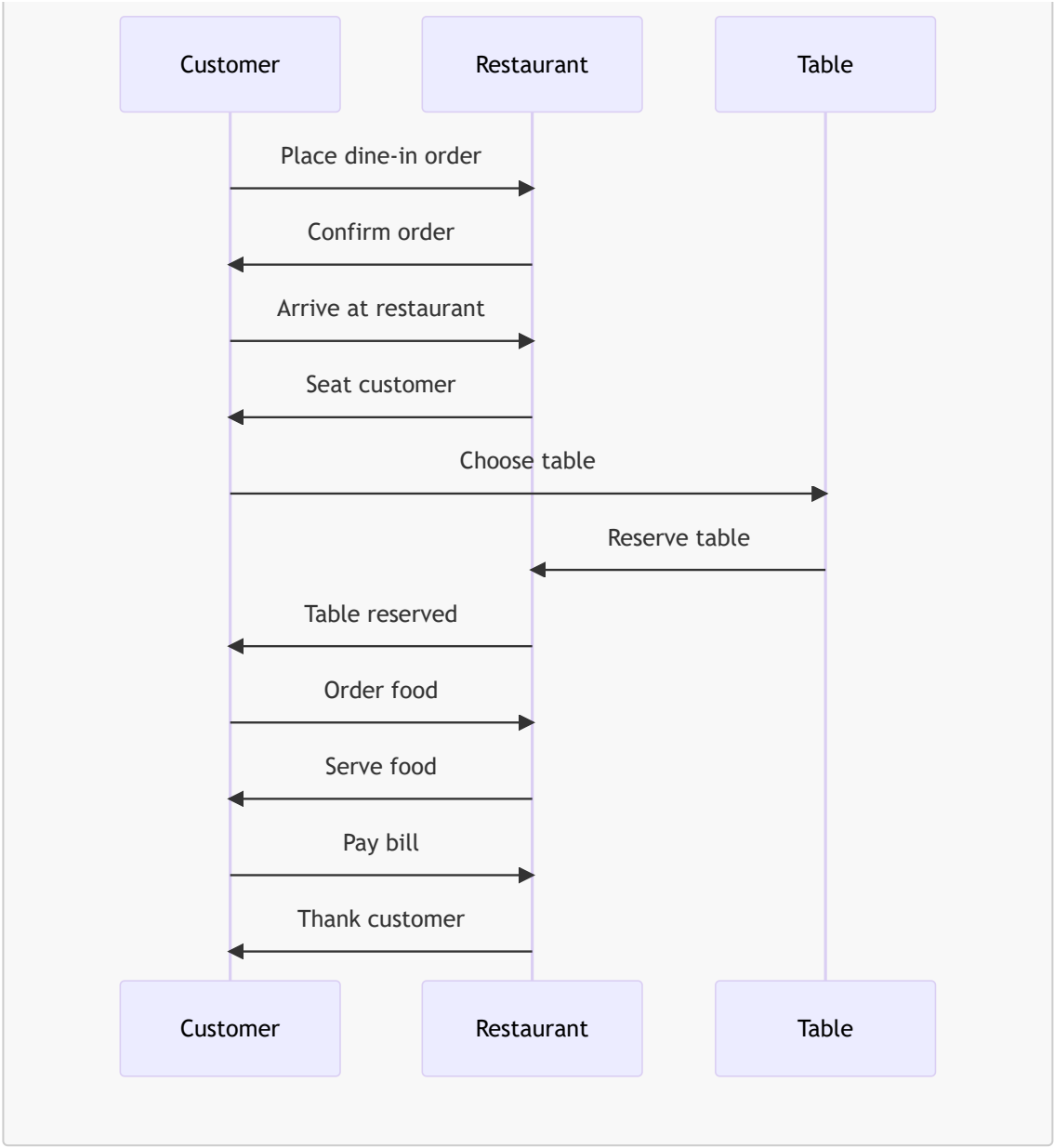
1.4.2.1. delivery order flow



1.4.2.2. takeaway order flow

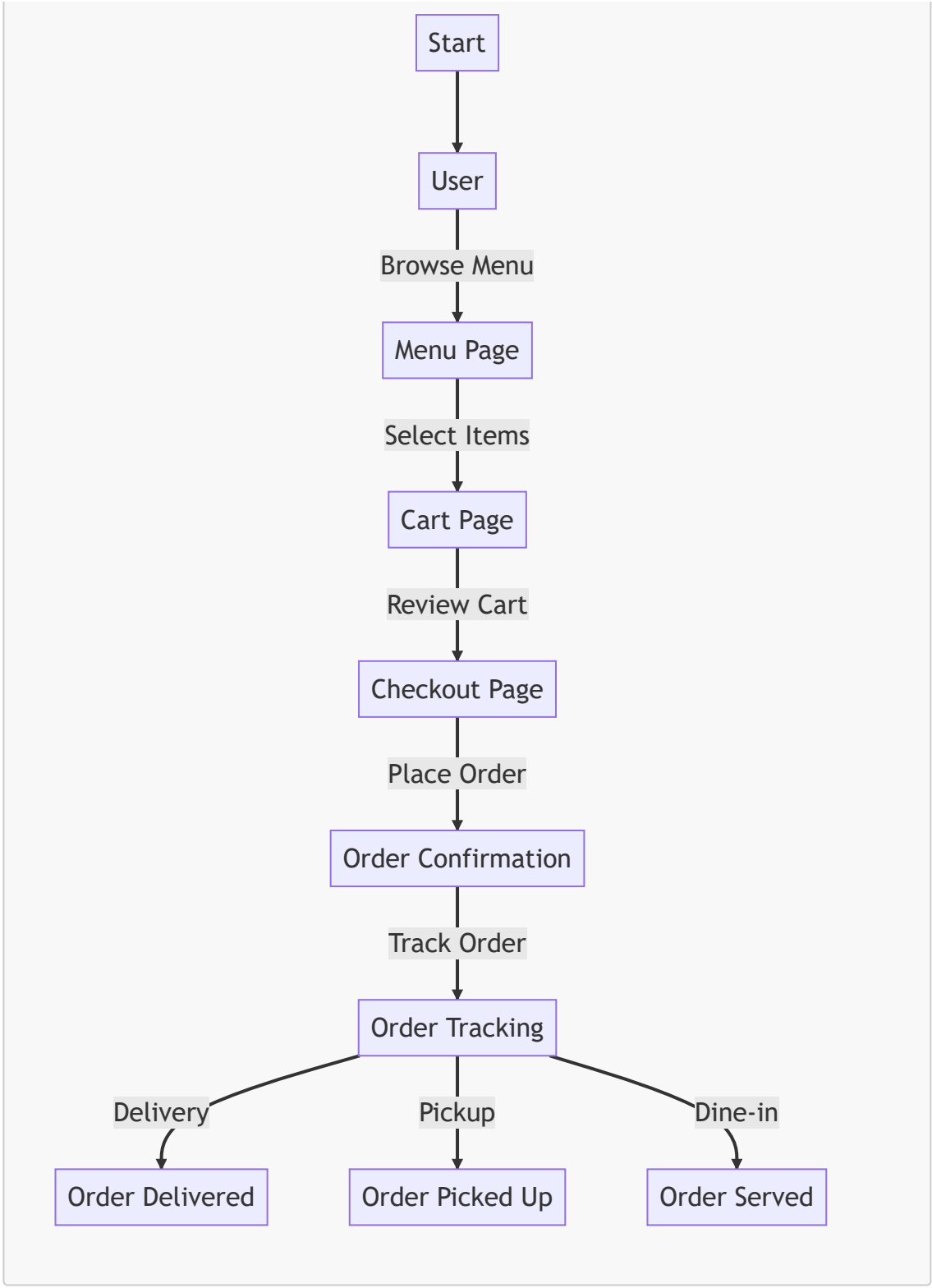


1.4.2.3. dine-in order flow

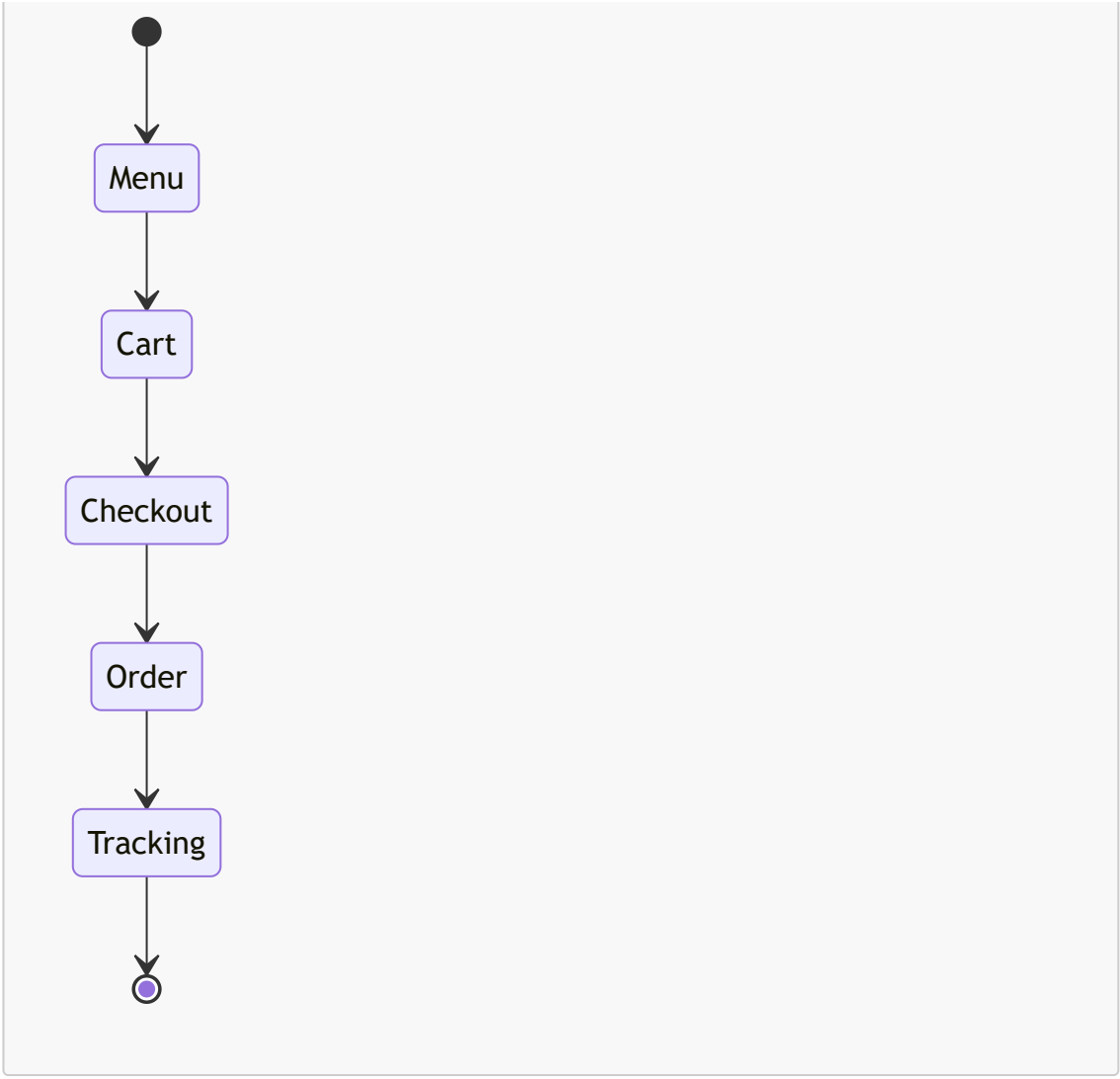


1.4.3. Use case diagrams

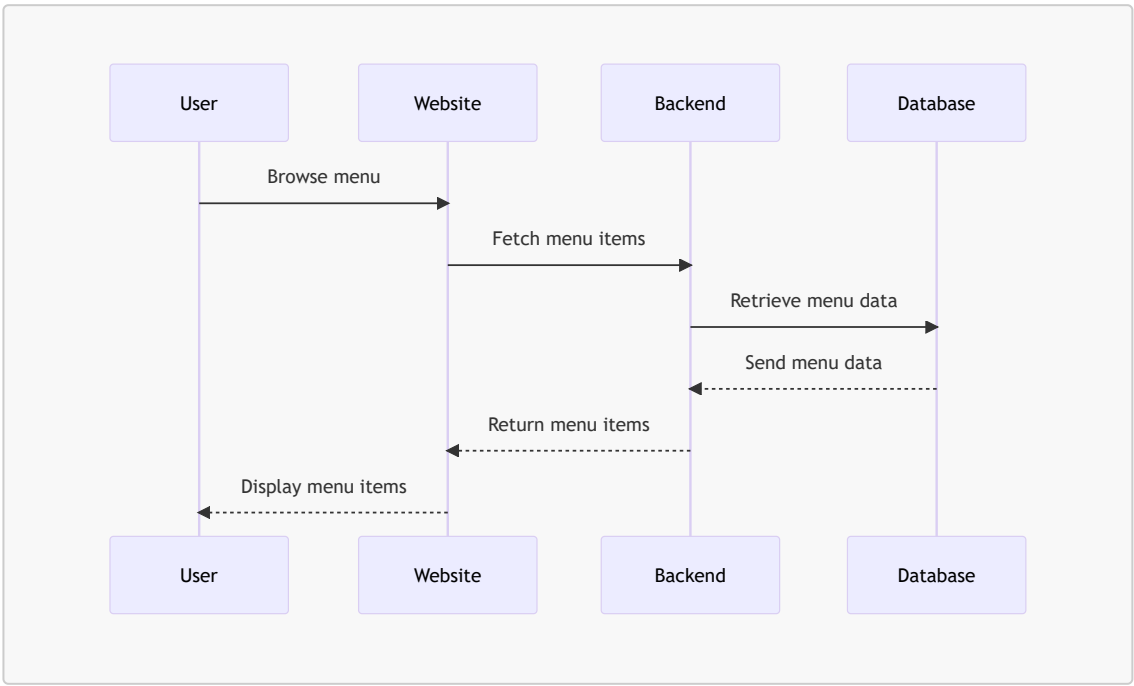
1.4.3.1. Activity diagrams



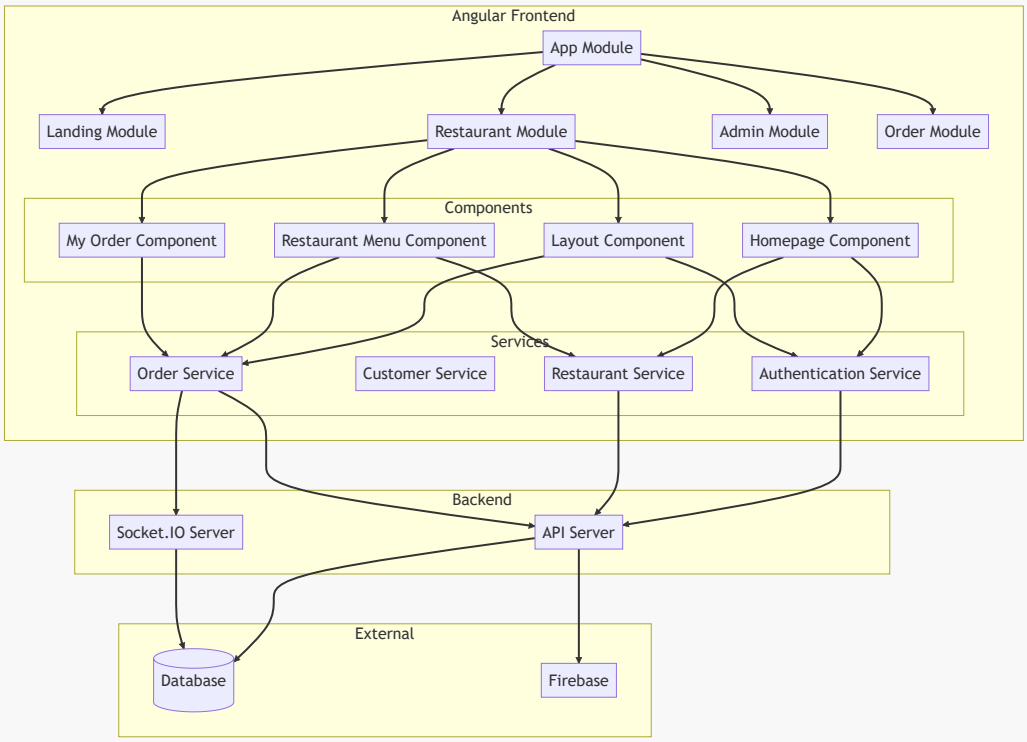
1.4.3.2. State machine diagrams



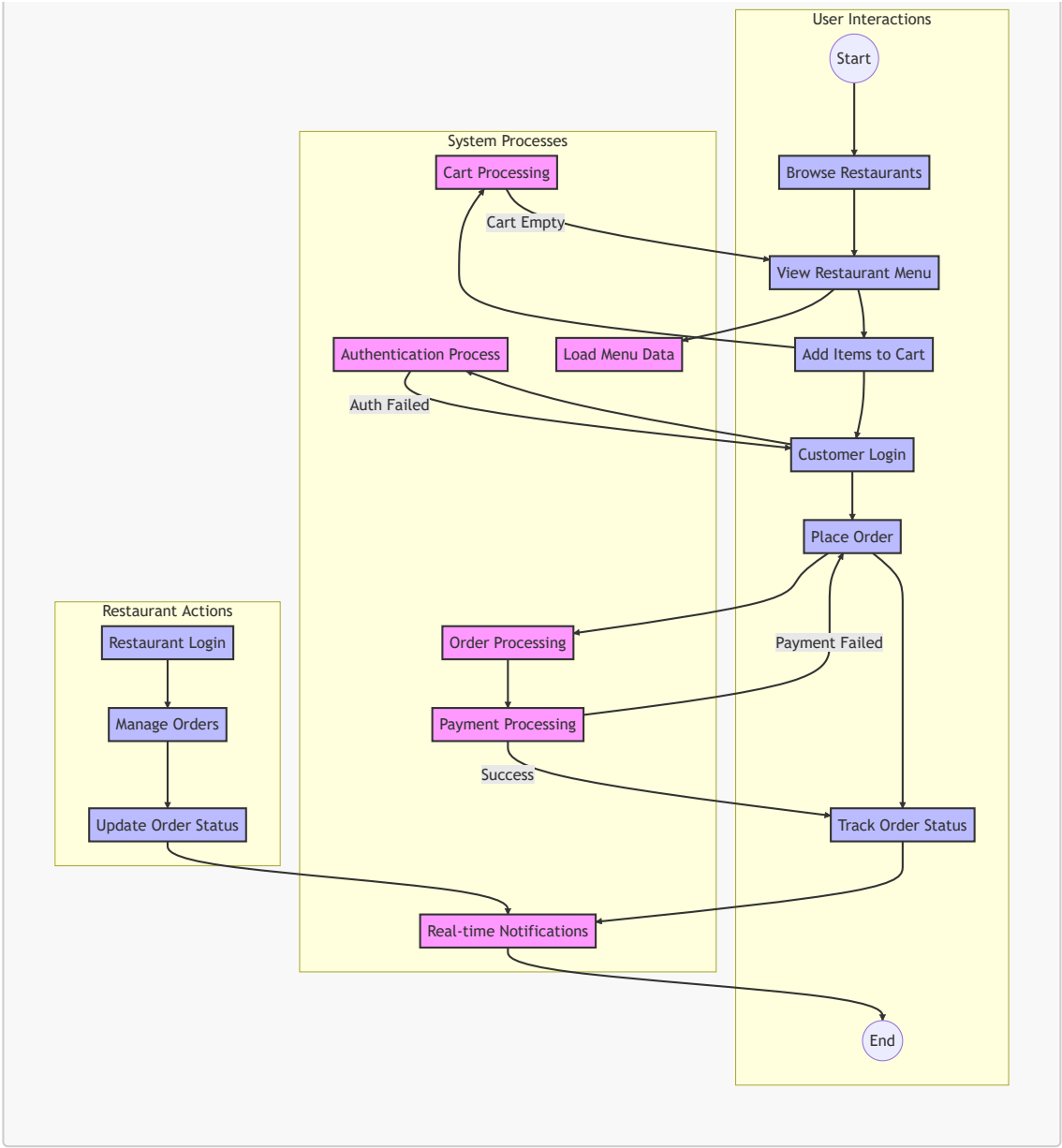
1.4.3.3. Sequence diagrams



1.4.3.4. Communication diagrams



1.4.3.5. Interaction overview diagrams



1.4.4. Key Use Cases and Scenarios

1. **User Registration and Login**
    - Users can create an account using their phonenumber.
    - Users can log in to their account to access personalized features.
2. **Browsing Restaurant Listings**
    - Users can browse a list of available restaurants based on their location.
    - Users can view restaurant details, including menus, reviews, and ratings.
3. **Placing an Order**
    - Users can select items from a restaurant's menu and add them to their cart.
    - Users can customize their order with special instructions or preferences.
    - Users can choose between delivery, pickup, or dine-in options.
4. **Order Tracking**
    - Users can track the status of their order in real-time.

    - Users receive notifications about order updates, including preparation, delivery, and completion.

5. **Payment Processing**
    - Users can pay for their order using various payment methods, including credit/debit cards, digital wallets, and UPI.
    - Users receive a confirmation of their payment and order details.

6. **User Reviews and Ratings**
    - Users can leave reviews and ratings for restaurants they have ordered from.
    - Users can read reviews and ratings from other customers to make informed decisions.

7. **Customer Support**
    - Users can contact customer support for assistance with their orders.
    - Users can report issues or provide feedback about their experience.

8. **Promotions and Discounts**
    - Users can apply promotional codes or discounts to their orders.
    - Users receive notifications about special offers and promotions.

9. **Account Management**
    - Users can update their account information, including contact details and payment methods.
    - Users can manage their notification preferences and privacy settings.

1.5. Technical Architecture

1.5.1. Technology Stack Overview

The website is built using the following technologies:

- Frontend: Angular - angular is a platform and framework for building single-page client applications using HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.
- Backend: Node.js - Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser.
- Database: MongoDB - MongoDB is a general-purpose, document-based, distributed database built for modern application developers and for the cloud era.
- Hosting: firebase.com - Firebase is a platform developed by Google for creating mobile and web applications. It was originally an independent company founded in 2011. In 2014, Google acquired the platform and it is now their flagship offering for app development.

- Payment Gateway: razorpay - Razorpay is a payment gateway that allows businesses to accept, process, and disburse payments with its product suite.
- Messaging Service: whatsapp - WhatsApp is a messaging service that allows users to send text messages, voice messages, images, and videos over the internet.
- Other Tools: Git, Postman, VS Code - Git is a distributed version control system for tracking changes in source code during software development. Postman is a collaboration platform for API development that allows users to design, mock, document, monitor, and test APIs. VS Code is a source-code editor developed by Microsoft for Windows, Linux, and macOS.

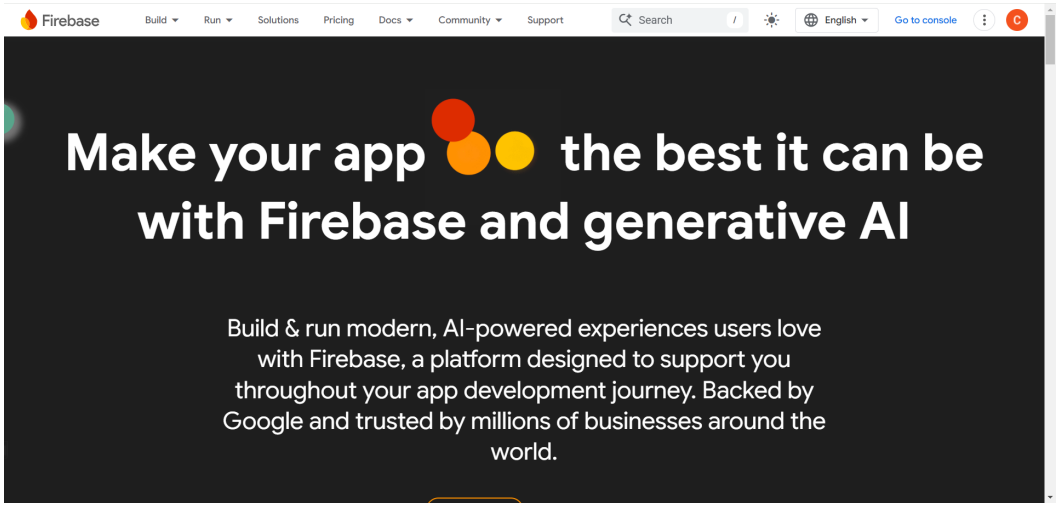
1.5.2. High-Level Architecture Diagram

1.5.3. Deployment and Hosting Details

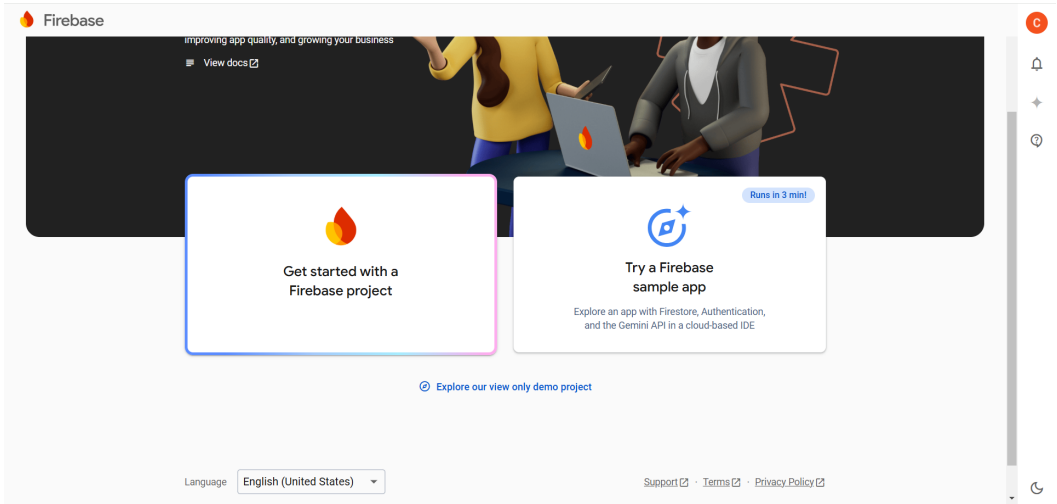
The website is deployed on firebase.com and hosted on Google Cloud Platform. The deployment process involves building the Angular frontend and deploying it to firebase hosting. The backend is deployed as a Node.js application on firebase functions. The database is hosted on MongoDB Atlas.

1.5.4. Environment Setup

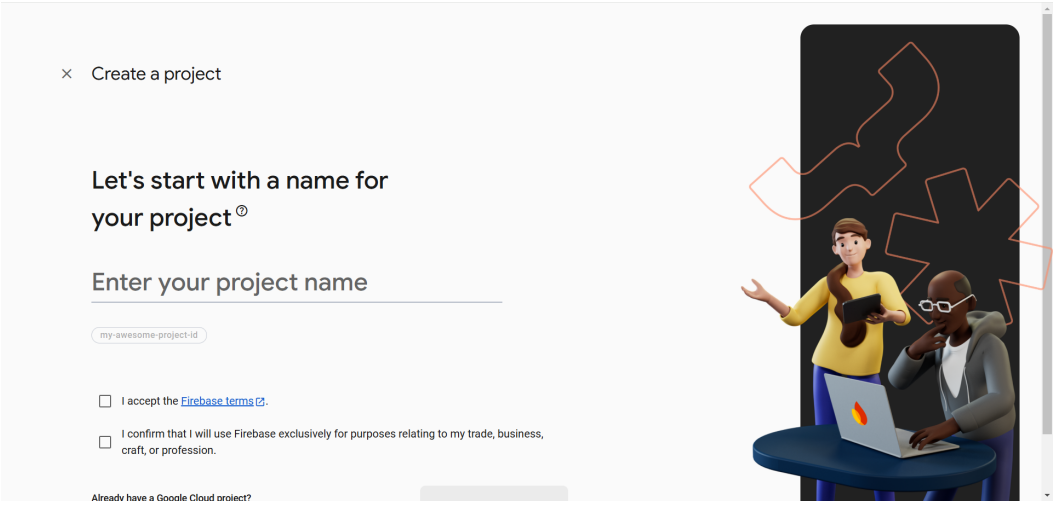
1. Go to `https://firebase.google.com/` and click on `Go to console`.



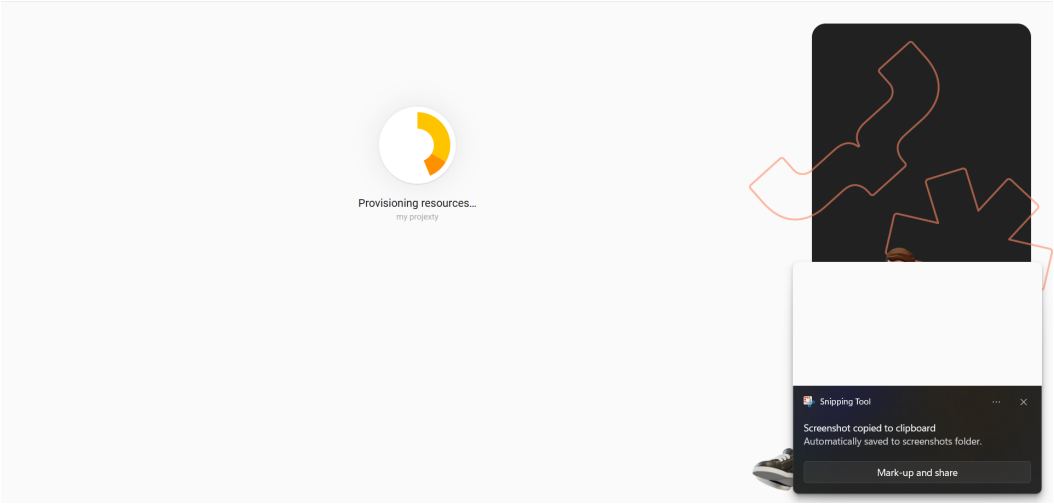
2. Click on `Get started with a Firebase project`.



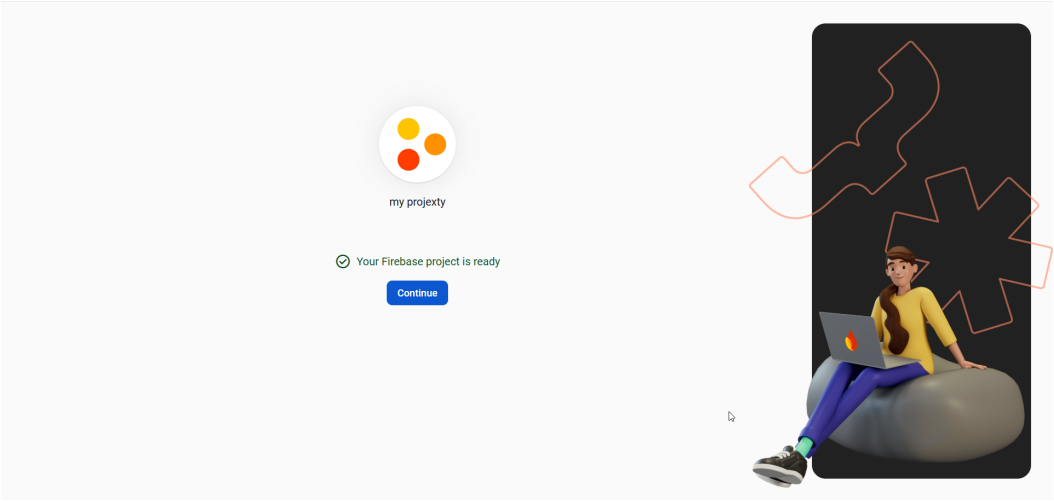
3. Enter the project name and click on `Continue`.



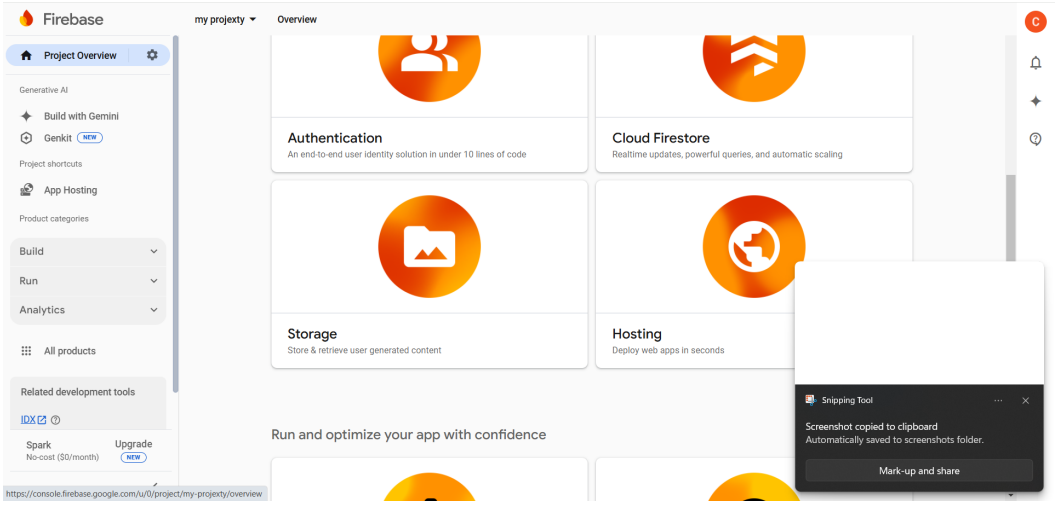
4. Click on `Create project`.



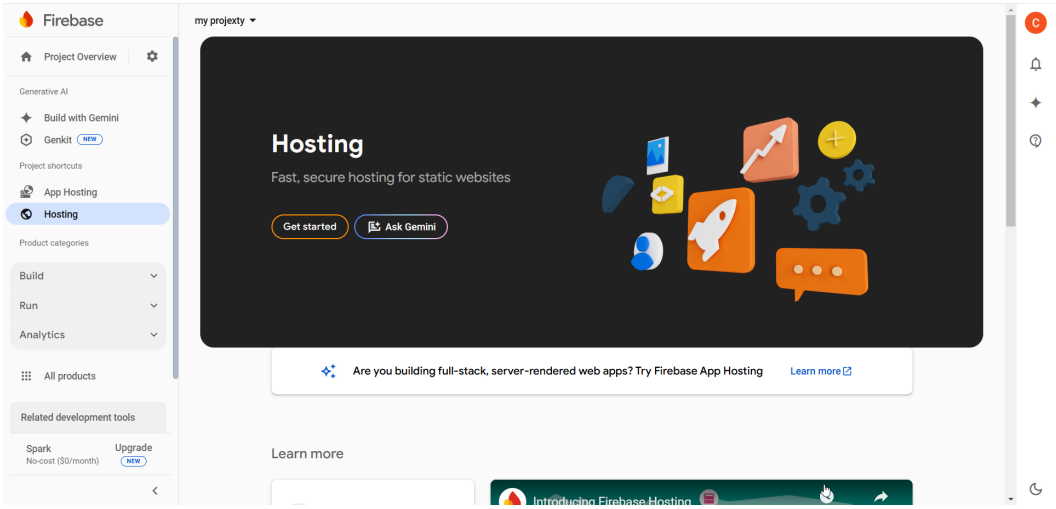
5. Click on `Continue`.



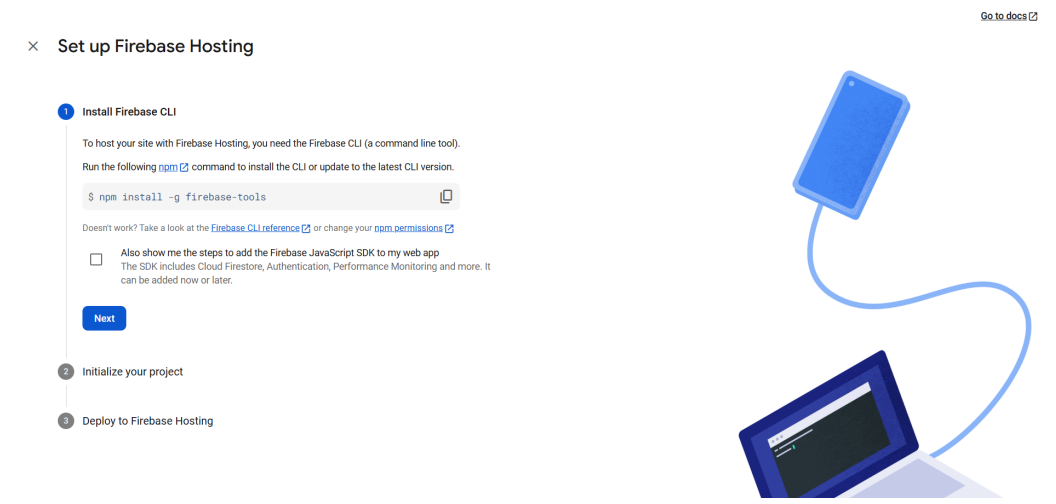
6. Click on "hosting".



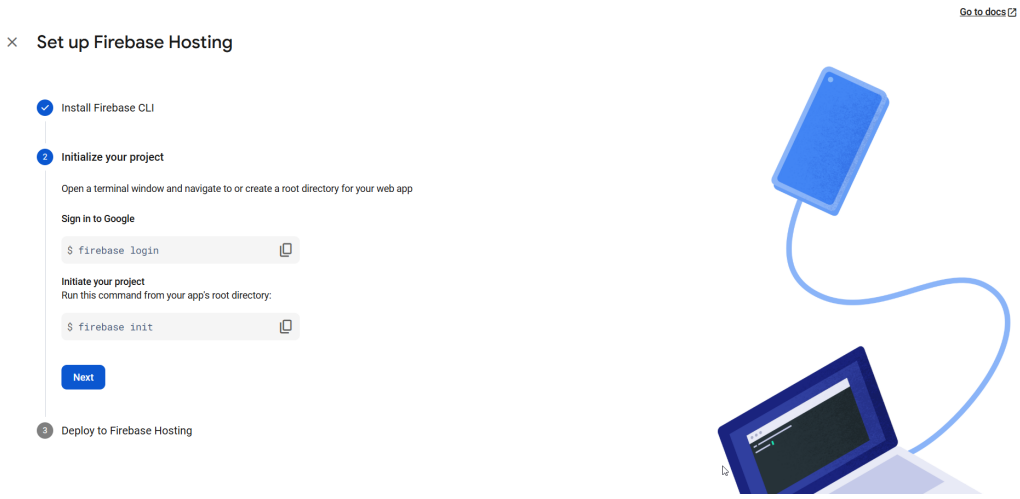
7. Click on `Get started`.



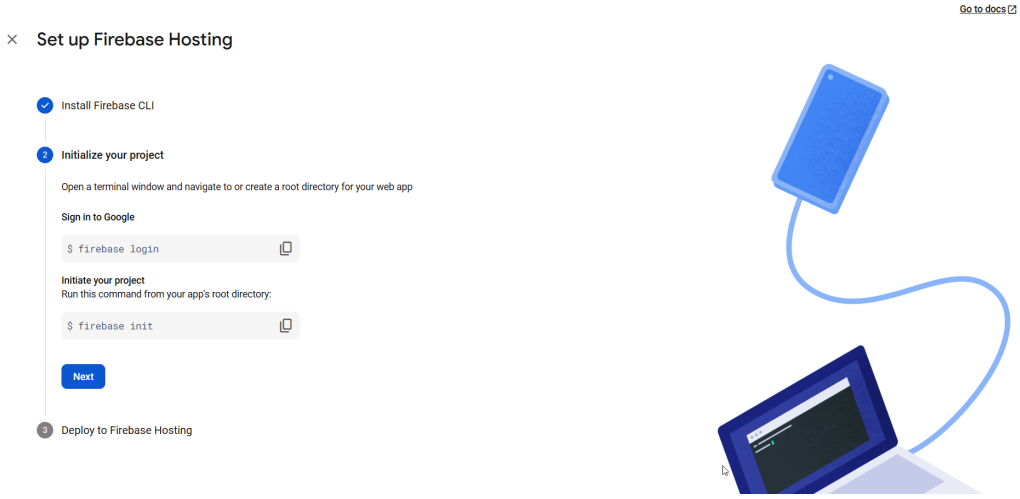
8. Install firebase tools using `npm install -g firebase-tools`.



9. Login to firebase using `firebase login`.



10. Initialize firebase using `firebase init`.



11. Deploy to Firebase Hosting using `firebase deploy`.

1.6. Beginner's Guide to Programming

1.6.1. Introduction to Web Development Basics

Web development is the process of building websites and web applications using a combination of HTML, CSS, and JavaScript. HTML is used to create the structure of a web page, CSS is used to style the page, and JavaScript is used to add interactivity and dynamic

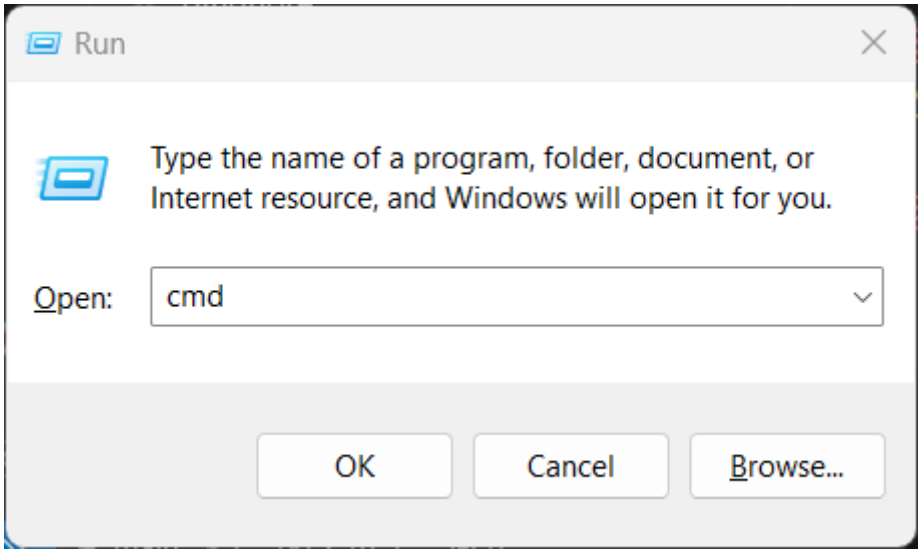
behavior to the page. Web development also involves working with backend technologies like Node.js and databases like MongoDB to create full-stack applications.

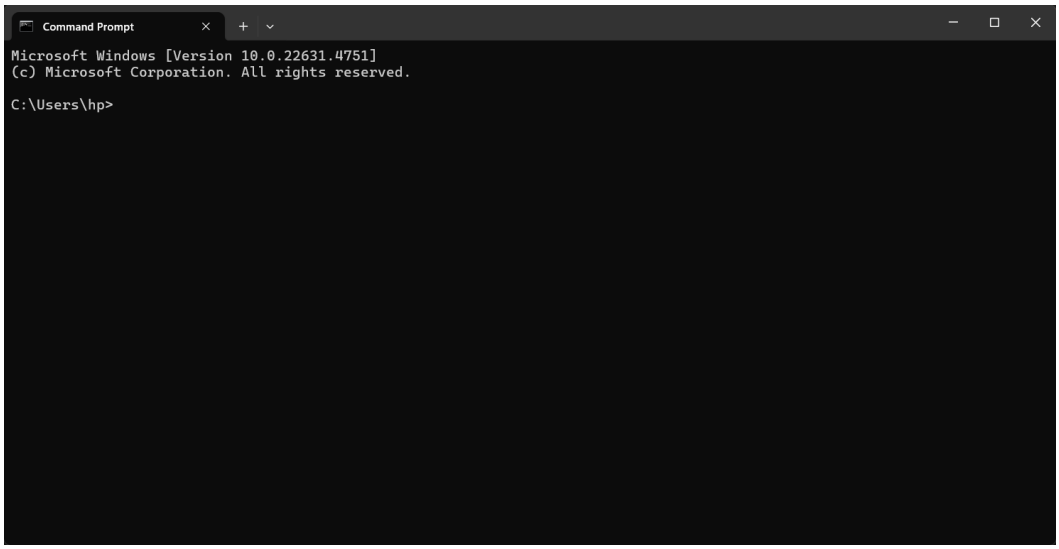
1.6.2. Overview of Tools and Software to Install

1. Node.js - Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. You can download Node.js from the official website and install it on your machine.
2. Angular CLI - The Angular CLI is a command-line interface tool that you use to initialize, develop, scaffold, and maintain Angular applications directly from a command shell.
3. MongoDB - MongoDB is a general-purpose, document-based, distributed database built for modern application developers and for the cloud era. You can download MongoDB from the official website and install it on your machine.
4. Git - Git is a distributed version control system for tracking changes in source code during software development. You can download Git from the official website and install it on your machine.
5. Postman - Postman is a collaboration platform for API development that allows users to design, mock, document, monitor, and test APIs. You can download Postman from the official website and install it on your machine.
6. VS Code - Visual Studio Code is a source-code editor developed by Microsoft for Windows, Linux, and macOS. You can download VS Code from the official website and install it on your machine.
7. Firebase CLI - The Firebase Command Line Interface (CLI) provides a variety of tools for managing, viewing, and deploying to Firebase projects. You can install the Firebase CLI using npm.
8. Angular Material - Angular Material is a UI component library for Angular that provides a set of high-quality UI components built with Angular and TypeScript. You can install Angular Material using npm.
9. Razorpay - Razorpay is a payment gateway that allows businesses to accept, process, and disburse payments with its product suite. You can sign up for a Razorpay account and get API keys to integrate with your application.
10. WhatsApp Business API - The WhatsApp Business API allows businesses to communicate with customers over WhatsApp. You can sign up for a WhatsApp Business API account and get API credentials to send messages.

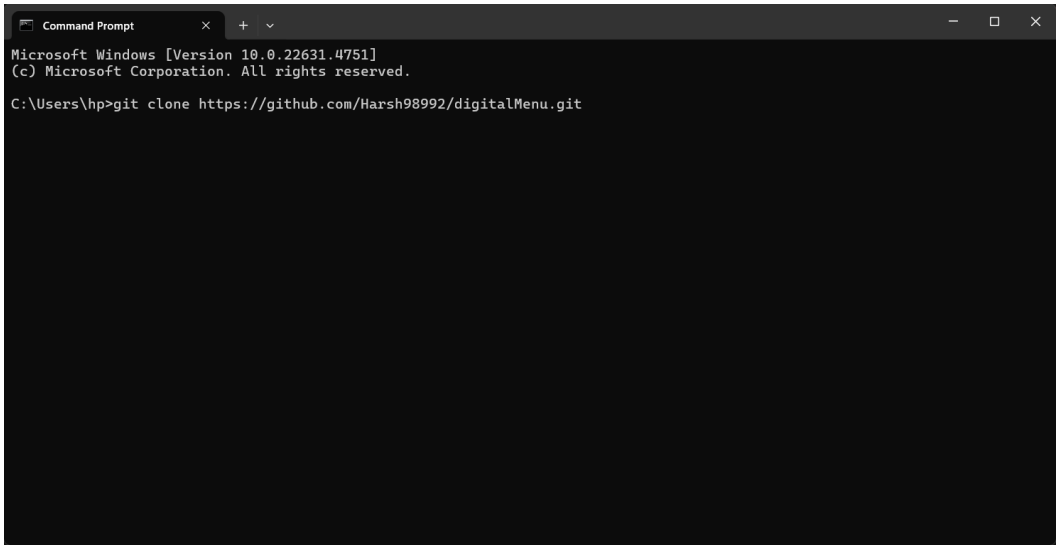
1.6.3. Step-by-Step Guide to Setting Up the Project Locally

1. open command prompt by pressing `windows + r` and typing `cmd` and press enter.

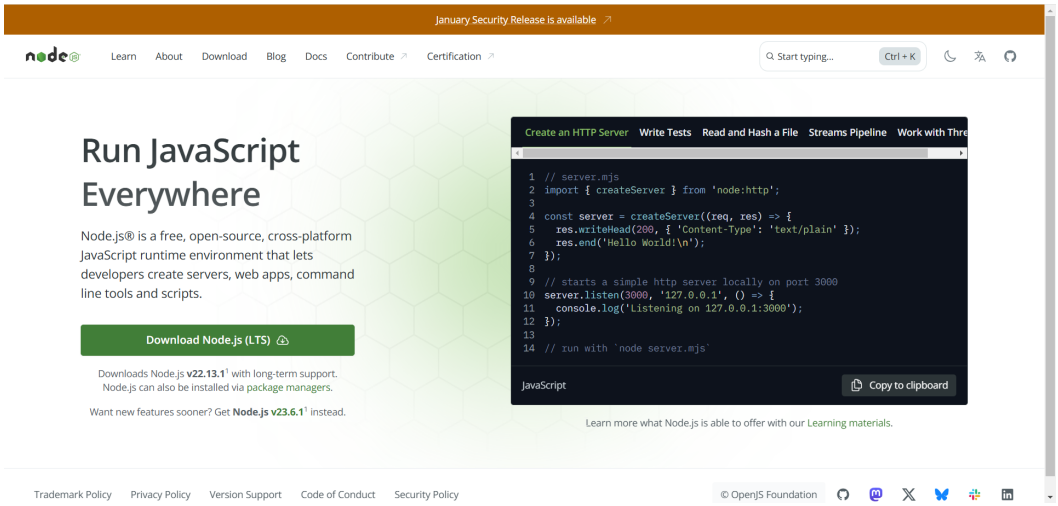




2. Clone the repository from GitHub using the `git clone https://github.com/Harsh98992/digitalMenu.git` command.



3. Install Node.js from the official website `https://nodejs.org/en/`.



4. Install Angular CLI using the `npm install -g @angular/cli` command.
5. Install MongoDB from the official website `https://www.mongodb.com/try/download/community`.
6. Install Git from the official website `https://git-scm.com/`.
7. Install Postman from the official website `https://www.postman.com/`.
8. Install VS Code from the official website `https://code.visualstudio.com/`.
9. Install Firebase CLI using the `npm install -g firebase-tools` command.

1.6.4. Suggested Learning Path

If you are new to web development, here is a suggested learning path to get started:

1. **HTML, CSS, and JavaScript Basics**
    - Learn the fundamentals of HTML from the `https://www.w3schools.com/html/` website.
    - Learn the basics of CSS from the `https://www.w3schools.com/css/` website.
    - Learn JavaScript basics from the `https://www.w3schools.com/js/` website.
    - Practice building simple web pages using HTML, CSS, and JavaScript.
2. **Angular Basics**
    - Learn the basics of Angular from the `https://angular.io/docs` website.
    - Build a simple Angular application using components, services, and modules.
    - Learn how to use Angular CLI to scaffold and generate code.
3. **Node.js Basics**
    - Dive into Node.js basics from the Node.js documentation.
    - Build a simple backend application using Express.js.
    - Explore how to handle routing, middleware, and RESTful API endpoints.
4. **MongoDB Basics**
    - Learn MongoDB basics from the MongoDB documentation.

- Set up a MongoDB database and connect it with your Node.js backend application.
- Practice performing CRUD operations (Create, Read, Update, Delete).

5. **Deployment to Firebase Hosting**
    - Learn how to deploy your application using Firebase Hosting.
    - Set up Firebase CLI and configure your project for deployment.
6. **Payment Gateway Integration**
    - Integrate a payment gateway like Razorpay. Check out the Razorpay Documentation.
    - Implement the necessary steps to handle transactions and payments in your application.
7. **Messaging Service Integration**
    - Learn how to integrate a messaging service like WhatsApp using the WhatsApp Business API.
    - Set up the API and configure messaging functionalities.
8. **Testing and Debugging**
    - Learn how to test and debug your application using tools like Jest for unit testing and Postman for API testing.
    - Implement end-to-end testing to ensure the stability of your application.

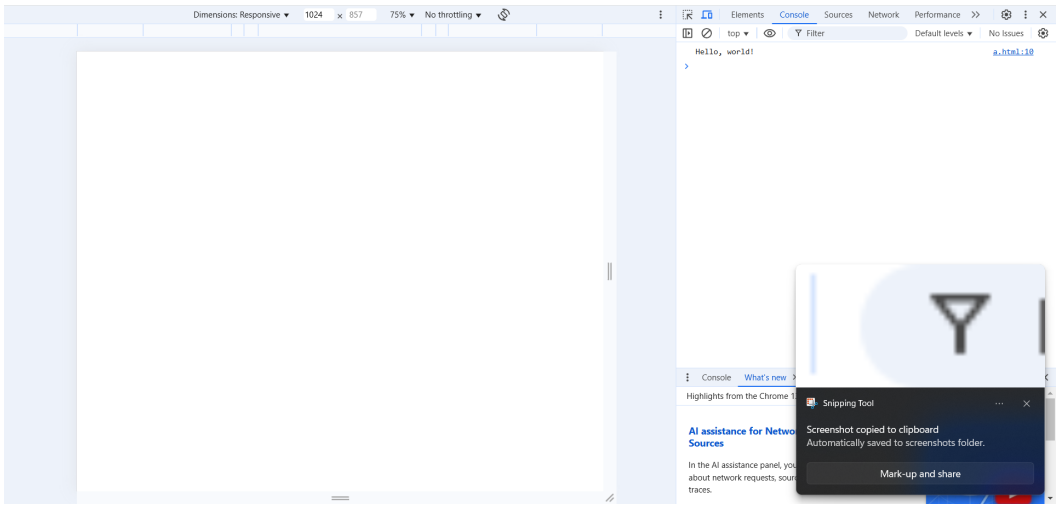
1.6.5. Debugging Basics

Debugging is the process of finding and fixing errors in your code. Here are some basic debugging techniques:

1. Use console.log() statements to print values and debug information.

```
ngOnInit(): void {
    this.bnIdle
        .startWatching(this.idleDuration)
        .subscribe((isTimedOut: boolean) => {
            if (isTimedOut) {
                console.log("session expired")
                this.refreshPage();
            }
        });
}
```

2. Use the browser developer tools to inspect elements, view console logs, and debug JavaScript code.



3. Use breakpoints in your code to pause execution and inspect variables.
4. Use the Angular CLI to run the project in development mode and view error messages in the console.
5. Use the Postman tool to test APIs and view response data.
6. Use the VS Code debugger to step through your code and inspect variables.
7. Use the Firebase CLI to view logs and debug cloud functions.

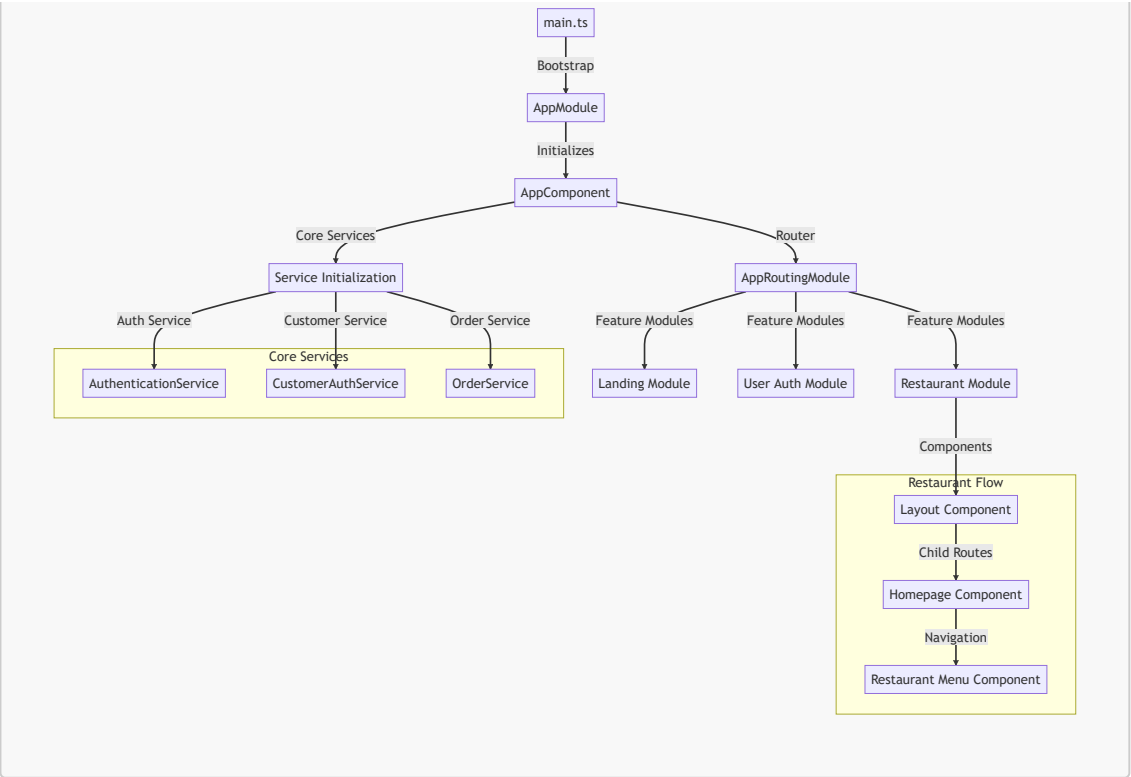
1.7. Codebase Structure and Flow

1.7.1. Overview of the Codebase

The codebase is organized into the following directories:

- `src` - Contains the source code for the Angular frontend application.
- `src/app` - Contains the Angular components, services, and modules.
- `src/assets` - Contains static assets like images, fonts, and stylesheets.
- `src/api` - Contains API endpoints and services for interacting with the backend.

1.7.2. Code Execution Flow



1.7.3. Understanding Functions and Modules

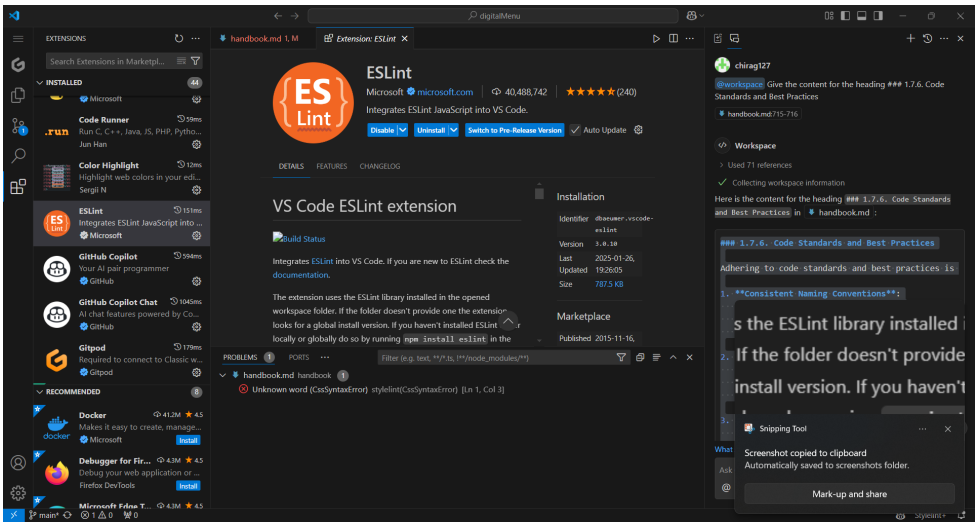
1.7.4. Step-by-Step Explanation of a Key Feature

1.7.5. Reading the Code

1.7.6. Code Standards and Best Practices

Adhering to code standards and best practices is crucial for maintaining a high-quality codebase. Below are some guidelines to follow:

1. **Consistent Naming Conventions**:
    - Use camelCase for variables and functions.
    - Use PascalCase for classes and components.
    - Use UPPER_SNAKE_CASE for constants.
2. **Code Formatting**:
    - Use a eslint to enforce consistent code formatting.

- 

- Indent code blocks with 2 spaces.
- Limit lines to 80 characters.

3. **Commenting and Documentation**:
    - Write clear and concise comments for complex logic.
    - Use JSDoc or similar tools for documenting functions and classes.
    - Update documentation regularly to reflect code changes.
4. **Error Handling**:
    - Use try-catch blocks to handle exceptions.
    - Log errors for debugging purposes.
    - Provide meaningful error messages to users.
5. **Code Reviews**:
    - Conduct regular code reviews to ensure code quality.
    - Provide constructive feedback during code reviews.
    - Address code review comments promptly.
6. **Testing**:
    - Write unit tests for all functions and components.
    - Use test-driven development (TDD) where applicable.
    - Ensure tests cover edge cases and potential failure points.
7. **Version Control**:
    - Use meaningful commit messages.
    - Commit code frequently with small, incremental changes.
    - Use branches for new features and bug fixes.
8. **Performance Optimization**:
    - Optimize code for performance where necessary.
    - Avoid premature optimization; focus on readability and maintainability first.
    - Profile and benchmark code to identify performance bottlenecks.

9. **Security**:
    - Follow best practices for securing code and data.
    - Validate and sanitize user inputs.
    - Use secure coding practices to prevent vulnerabilities.
10. **Continuous Integration and Deployment (CI/CD)**:
    - Use CI/CD pipelines to automate testing and deployment.
    - Ensure that the build process is reliable and repeatable.
    - Monitor deployments and rollback if issues are detected.

1.8. API Documentation

1.8.1. Overview of API Usage and Purpose

The digital menu system's API infrastructure serves as the backbone of communication between the frontend application and backend services. Our APIs are built using RESTful principles and are primarily used for:

1. **Menu Management**
    - Retrieving restaurant menus and item details
    - Managing menu items, categories, and pricing
    - Handling menu availability and special offers
2. **Order Processing**
    - Creating and managing customer orders
    - Tracking order status and updates
    - Managing delivery/pickup preferences
3. **User Management**
    - Customer authentication and authorization
    - Profile management and preferences
    - Order history and favorites
4. **Restaurant Operations**
    - Staff authentication and role-based access
    - Order queue management
    - Real-time kitchen notifications

All APIs use JSON for data exchange and require proper authentication using JWT tokens. The base URL for all API endpoints is `/api/v1`, and requests are secured using HTTPS protocol.

1.8.2. API Endpoint List

1.8.2.1. Admin Panel endpoints

1.8.2.1.1. Get Restaurants by Status

- **Endpoint**: `/api/v1/admin/getRestaurantsByStatus/:restaurantVerified`
- **Method**: GET
- **Description**: Retrieves a list of restaurants based on their verification status.
- **Parameters**:
  - `restaurantVerified` (boolean): Indicates whether the restaurant is verified or not.
  - Example: `/api/v1/admin/getRestaurantsByStatus/true`
  - Example Response:

```
{
    "restaurants": [
        {
            "id": "123",
            "name": "Restaurant A",
            "verified": true
        },
        {
            "id": "456",
            "name": "Restaurant B",
            "verified": true
        }
    ]
}
```

- **Response**: Returns an array of restaurant objects with their details.
- **Authorization**: Admin role required.
- **Error Handling**: Returns an error message if the request fails.
- **Sample Code**:

```
getRestaurantsByStatus(restaurantVerified: boolean) {
    return this.http.get(

`${this.apiUrl}/v1/admin/getRestaurantsByStatus/${restaurantV
erified}`
    );
}
```

1.8.2.1.2. Get Restaurant Payment Details

- **Endpoint**: `/api/v1/payment/getAccountPaymentDetails`

- **Method**: GET
- **Description**: Fetches payment details for all restaurant accounts.
- **Parameters**: None.
- **Example Response**:

```
{
    "payments": [
        {
            "restaurantId": "123",
            "restaurantName": "Restaurant A",
            "totalEarnings": 15000,
            "pendingAmount": 5000,
            "lastPaymentDate": "2025-01-10"
        },
        {
            "restaurantId": "456",
            "restaurantName": "Restaurant B",
            "totalEarnings": 20000,
            "pendingAmount": 3000,
            "lastPaymentDate": "2025-01-12"
        }
    ]
}
```

- **Response**: Returns an array of objects containing payment information for each restaurant.
- **Authorization**: Admin role required.
- **Error Handling**: Returns an error message if the request fails.
- **Sample Code**:

```
getRestaurantPayment() {
    return this.http.get(
        `${this.apiUrl}/v1/payment/getAccountPaymentDetails`
    );
}
```

1.8.2.1.3. Get Account Transfer Details

- **Endpoint**: `/api/v1/payment/getAccountTransferDetails/:orderId`
- **Method**: GET
- **Description**: Retrieves transfer details for a specific order.

- **Parameters**:
    - `orderId` (string): The unique identifier for the order.
- **Example**: `/api/v1/payment/getAccountTransferDetails/ORD12345`
- **Example Response**:

```
{
    "orderId": "ORD12345",
    "restaurantId": "123",
    "transferAmount": 2000,
    "transferDate": "2025-01-15",
    "status": "Completed"
}
```

- **Response**: Returns an object with the details of the account transfer related to the order.
- **Authorization**: Admin role required.
- **Error Handling**: Returns an error message if the order ID is invalid or the request fails.
- **Sample Code**:

```
getAccountTransferDetails(orderId: string) {
    return this.http.get(

`${this.apiUrl}/v1/payment/getAccountTransferDetails/${orderI
d}`
    );
}
```

1.8.2.1.4. Get Admin Restaurant Data

- **Endpoint**: `/api/v1/admin/getRestaurantDetail/:id`
- **Method**: GET
- **Description**: Fetches detailed information about a specific restaurant.
- **Parameters**:
    - `id` (string): The unique identifier for the restaurant.
- **Example**: `/api/v1/admin/getRestaurantDetail/123`
- **Example Response**:

```
{
    "id": "123",
    "name": "Restaurant A",
    "verified": true,
    "owner": "John Doe",
    "contact": "123-456-7890",
    "address": "123 Main St, City, State",
    "cuisine": ["Italian", "Mexican"],
    "ratings": 4.5
}
```

- **Response**: Returns detailed information about the restaurant, including owner details, address, and ratings.
- **Authorization**: Admin role required.
- **Error Handling**: Returns an error message if the restaurant ID is invalid or the request fails.
- **Sample Code**:

```
getAdminRestaurantData(id: string) {
    return this.http.get(
        `${this.apiUrl}/v1/admin/getRestaurantDetail/${id}`
    );
}
```

1.8.2.1.5. Change Restaurant Status

- **Endpoint**: `/api/v1/admin/changeRestaurantStatus/:id`
- **Method**: PATCH
- **Description**: Updates the verification status of a restaurant.
- **Parameters**:
    - `id` (string): The unique identifier for the restaurant.
    - Request Body:

```
{
    "verified": true
}
```

- **Example**: `/api/v1/admin/changeRestaurantStatus/123`
- **Example Response**:

```
{
    "message": "Restaurant status updated successfully."
}
```

- **Response**: Returns a success message upon updating the restaurant's status.
- **Authorization**: Admin role required.
- **Error Handling**: Returns an error message if the restaurant ID is invalid or the request fails.
- **Sample Code**:

```
changeRestaurantStatus(id: string, data: any) {
    return this.http.patch(

`${this.apiUrl}/v1/admin/changeRestaurantStatus/${id}`,
        data
    );
}
```

1.8.2.1.6. Edit Restaurant Details

- **Endpoint**: `/api/v1/admin/editRestaurant/:id`
- **Method**: PATCH
- **Description**: Updates the details of a restaurant.
- **Parameters**:
    - `id` (string): The unique identifier for the restaurant.
    - Request Body:

```
{
    "name": "New Restaurant Name",
    "contact": "987-654-3210",
    "address": "456 Main St, City, State",
    "cuisine": ["Indian", "Chinese"]
}
```

- **Example**: `/api/v1/admin/editRestaurant/123`
- **Example Response**:

```
```

```
{
    "message": "Restaurant details updated successfully."
}
```

- **Response**: Returns a success message upon updating the restaurant's details.
- **Authorization**: Admin role required.
- **Error Handling**: Returns an error message if the restaurant ID is invalid or the request fails.
- **Sample Code**:

```
editRestaurant(id: string, data: any) {
    return this.http.patch(
        `${this.apiUrl}/v1/admin/editRestaurant/${id}`,
        data
    );
}
```

1.8.2.1.7. View All Users of a Restaurant

- **Endpoint**: `/api/v1/admin/viewAllUsersOfRestaurant/:id`
- **Method**: GET
- **Description**: Retrieves a list of all users associated with a specific restaurant.
- **Parameters**:
    - `id` (string): The unique identifier for the restaurant.
    - Example: `/api/v1/admin/viewAllUsersOfRestaurant/123`
    - Example Response:

```
{
    "users": [
        {
            "id": "456",
            "name": "User A",
            "email": "abc#example.com",
            "role": "Customer"
        },
        {
            "id": "789",
            "name": "User B",
            "email": "xyz#example.com",
            "role": "Staff"
        }
```

```
    ]
}
```

- **Response**: Returns an array of user objects with their details.
- **Authorization**: Admin role required.
- **Error Handling**: Returns an error message if the restaurant ID is invalid or the request fails.
- **Sample Code**:

```
viewAllUsersOfRestaurant(id: string) {
    return this.http.get(

`${this.apiUrl}/v1/admin/viewAllUsersOfRestaurant/${id}`
    );
}
```

1.8.2.1.8. Send Email to Restaurant

- **Endpoint**: `/api/v1/admin/sendEmailToRestaurant`
- **Method**: POST
- **Description**: Sends an email notification to a restaurant.
- **Request Body**:

```
{
    "restaurantId": "123",
    "subject": "Order Notification",
    "message": "You have a new order pending."
}
```

- **Response**: Returns a success message upon sending the email.
- **Authorization**: Admin role required.
- **Error Handling**: Returns an error message if the request fails.
- **Sample Code**:

```
sendEmailToRestaurant(data: any) {
    return this.http.post(
        `${this.apiUrl}/v1/admin/sendEmailToRestaurant`,
        data
```

```
        );
    }
```

1.8.2.1.9. Export JSON to Excel

- **Description**: Converts JSON data to an Excel file and downloads it.
- **Parameters**:
    - `jsonData` (array): The JSON data to export.
    - `fileName` (string): The name of the Excel file.
    - **Sample Code**:

```
        exportJsonToExcel(jsonData: any[], fileName: string): void
{
            const worksheet: XLSX.WorkSheet =
XLSX.utils.json_to_sheet(jsonData);
            const workbook: XLSX.WorkBook = {
                Sheets: { data: worksheet },
                SheetNames: ["data"],
            };
            const excelBuffer: any = XLSX.write(workbook, {
                bookType: "xlsx",
                type: "array",
            });
            this.saveAsExcelFile(excelBuffer, fileName);
        }

        private saveAsExcelFile(buffer: any, fileName: string):
void {
            const data: Blob = new Blob([buffer], { type:
this.EXCEL_TYPE });
            saveAs(
                data,
                fileName + "_export_" + new Date().getTime() +
this.EXCEL_EXTENSION
            );
        }
```

1.8.2.2. Authentication Endpoints

1.8.2.2.1. changePassword

- **Endpoint**: `/api/v1/user/updatePassword`
- **Method**: PATCH
- **Description**: Updates the user's password.

- **Parameters**:
    - Request Body:

```
{
    "oldPassword": "password123",
    "newPassword": "newpassword123"
}
```

- **Response**: Returns a success message upon updating the password.
- **Authorization**: User authentication required.
- **Error Handling**: Returns an error message if the request fails.
- **Sample Code**:

```
changePassword(requestData) {
    return this.http.patch(
        `${this.apiUrl}/v1/user/updatePassword`,
        requestData
    );
}
```

1.8.2.2.2. resetPassword

- **Endpoint**: `/api/v1/user/resetPassword/:token`
- **Method**: PATCH
- **Description**: Resets the user's password using a valid reset token.
- **Parameters**:
    - `token` (string): A unique token sent to the user's email for password reset.
    - Request Body:

```
{
    "password": "newpassword123"
}
```

- **Response**: Returns a success message upon successfully resetting the password.
- **Authorization**: No authentication required; token-based validation.
- **Error Handling**: Returns an error if the token is invalid, expired, or the request fails.
- **Sample Code**:

```
resetPassword(password: string, token: string) {
    return this.http.patch(
        `${this.apiUrl}/v1/user/resetPassword/${token}`,
        { password }
    );
}
```

1.8.2.2.3. register

- **Endpoint**: `/api/v1/user/signup`
- **Method**: POST
- **Description**: Registers a new user with the provided details.
- **Parameters**:
    - Request Body:

```
{
    "name": "John Doe",
    "email": "johndoe@example.com",
    "password": "password123",
    "confirmPassword": "password123"
}
```

- **Response**: Returns a success message and user details upon successful registration.
- **Authorization**: No authentication required.
- **Error Handling**: Returns an error if the email is already registered or the request fails.
- **Sample Code**:

```
register(userData) {
    return this.http.post(`${this.apiUrl}/v1/user/signup`,
userData);
}
```

1.8.2.2.4. login

- **Endpoint**: `/api/v1/user/login`
- **Method**: POST
- **Description**: Logs in a user using their email and password.
- **Parameters**:
    - Request Body:

```
```

```
{
    "email": "johndoe@example.com",
    "password": "password123"
}
```

- **Response**: Returns a success message, user details, and an authentication token upon successful login.
- **Authorization**: No authentication required.
- **Error Handling**: Returns an error if the credentials are invalid or the request fails.
- **Sample Code**:

```
login(userData: { email: string; password: string }) {
    this.customerAuth.removeToken();
    return this.http.post(`${this.apiUrl}/v1/user/login`,
userData);
}
```

1.8.2.2.5. forgotPassword

- **Endpoint**: `/api/v1/user/forgotPassword`
- **Method**: POST
- **Description**: Sends a password reset link to the user's email address.
- **Parameters**:
    - Request Body:

```
{
    "email": "johndoe@example.com"
}
```

- **Response**: Returns a success message confirming that the reset link has been sent.
- **Authorization**: No authentication required.
- **Error Handling**: Returns an error if the email is not registered or the request fails.
- **Sample Code**:

```
forgotPassword(email: string) {
    return
this.http.post(`${this.apiUrl}/v1/user/forgotPassword`, {
email });
}
```

1.8.2.2.6. sendEmailVerificationOtp

- **Endpoint**: `/api/v1/user/emailVerification`
- **Method**: POST
- **Description**: Sends an OTP to the user's email for email verification.
- **Parameters**:
  - Request Body:

```
{
    "email": "johndoe@example.com"
}
```

- **Response**: Returns a success message confirming that the OTP has been sent.
- **Authorization**: No authentication required.
- **Error Handling**: Returns an error if the email is invalid or the request fails.
- **Sample Code**:

```
sendEmailVerificationOtp(email: string) {
    const data = { email };
    return
this.http.post(`${this.apiUrl}/v1/user/emailVerification`,
data);
}
```

1.8.2.2.7. verifyEmailOtp

- **Endpoint**: `/api/v1/user/verifyEmailOtp`
- **Method**: PUT
- **Description**: Verifies the OTP sent to the user's email.
- **Parameters**:
  - Request Body:

```
{
    "otp": "123456",
    "email": "johndoe@example.com"
}
```

- **Response**: Returns a success message upon successful verification of the email.
- **Authorization**: No authentication required.

- **Error Handling**: Returns an error if the OTP is invalid or expired.
- **Sample Code**:

```
verifyEmailOtp(otp: string, email: string) {
    return
this.http.put(`${this.apiUrl}/v1/user/verifyEmailOtp`, {
        otp,
        email,
    });
}
```

1.8.2.2.8. Utility Methods

- **setUserToken**: Saves the user's authentication token to `sessionStorage`.

```
setUserToken(token: string) {
    sessionStorage.clear();
    const driver = this.utilService.getPrinterDriver();
    localStorage.clear();
    if (driver) {
        localStorage.setItem("printerDriver",
JSON.stringify(driver));
    }
    sessionStorage.setItem("authToken", token);
}
```

- **getUserToken**: Retrieves the user's authentication token from `sessionStorage`.

```
getUserToken() {
    return sessionStorage.getItem("authToken");
}
```

- **removeToken**: Clears authentication token and resets session/local storage.

```
removeToken() {
    sessionStorage.clear();
    const driver = this.utilService.getPrinterDriver();
    localStorage.clear();
    if (driver) {
        localStorage.setItem("printerDriver",
JSON.stringify(driver));
    }
}
```

1.8.2.3. Customer Details Endpoints

1.8.2.3.1. Store Customer Details

- **Description**: Stores the customer's name and phone number in local storage.
- **Parameters**:
  - `name` (string): The customer's name.
  - `phoneNumber` (string): The customer's phone number.
  - **Sample Code**:
  - ```
        storeCustomerDetails(name: string, phoneNumber: string): void {
            localStorage.setItem('customerName', name);
            localStorage.setItem('customerPhoneNumber', phoneNumber);
        }
        ```
- **Usage**:

```
customerDetailsService.storeCustomerDetails("John Doe", "123-456-7890");
```

1.8.2.3.2. Get Customer Details

- **Description**: Retrieves the customer's name and phone number from local storage.
- **Parameters**: None.
- **Response**: Returns an object with the customer's name and phone number.
- **Sample Code**:

```
getCustomerDetails(): { name: string, phoneNumber: string } {
    const name = localStorage.getItem('customerName');
    const phoneNumber = localStorage.getItem('customerPhoneNumber');
    return { name: name, phoneNumber: phoneNumber };
}
```

- **Usage**:
```

```
const customerDetails =
customerDetailsService.getCustomerDetails();
console.log(customerDetails);
```

1.8.2.4. Customer Service Endpoints

1.8.2.4.1. Get Customer

- **Endpoint**: `/api/v1/customer/getCustomer`
- **Method**: GET
- **Description**: Retrieves the details of the currently logged-in customer.
- **Parameters**: None.
- **Response**: Returns an object with the customer's details.
- **Authorization**: Customer authentication required.
- **Error Handling**: Returns an error message if the request fails.
- **Sample Code**:

```
getCustomer() {
    return
this.http.get(`${this.apiUrl}/v1/customer/getCustomer`);
}
```

- **Usage**:

```
customerService.getCustomer().subscribe((data) => {
    console.log(data);
});
```

1.8.2.4.2. Add Customer Address

- **Endpoint**: `/api/v1/customer/addCustomerAddress`
- **Method**: PATCH
- **Description**: Adds a new address for the currently logged-in customer.
- **Parameters**:
    - `data`: An object containing the customer's address information.

- Example structure:

```
{
    "addressLine1": "123 Main St",
    "addressLine2": "Apt 4B",
    "city": "Metropolis",
    "state": "NY",
    "postalCode": "12345",
    "country": "USA"
}
```

- **Response**: Returns a confirmation message if the address is added successfully.
- **Authorization**: Customer authentication required.
- **Error Handling**: Returns an error message if the address cannot be added (e.g., invalid data or server error).
- **Sample Code**:

```
addCustomerAddress(data) {
    return
this.http.patch(`${this.apiUrl}/v1/customer/addCustomerAddres
s`, data);
}
```

- **Usage**:

```
const addressData = {
    addressLine1: "123 Main St",
    addressLine2: "Apt 4B",
    city: "Metropolis",
    state: "NY",
    postalCode: "12345",
    country: "USA",
};

customerService.addCustomerAddress(addressData).subscribe((re
sponse) => {
    console.log("Address added successfully:", response);
});
```

1.8.2.4.3. Edit Customer Address

- **Endpoint**: `/api/v1/customer/editCustomerAddress`
- **Method**: PATCH

- **Description**: Edits an existing address of the currently logged-in customer.
- **Parameters**:
    - `data`: An object containing the updated address information. It should include an address identifier (like `addressId`) and the updated address fields.
    - Example structure:

```
{
    "addressId": "123",
    "addressLine1": "456 New St",
    "addressLine2": "Apt 7C",
    "city": "Gotham",
    "state": "NY",
    "postalCode": "67890",
    "country": "USA"
}
```

- **Response**: Returns a confirmation message if the address is successfully updated.
- **Authorization**: Customer authentication required.
- **Error Handling**: Returns an error message if the address update fails (e.g., invalid address ID, missing fields, or server error).
- **Sample Code**:

```
editCustomerAddress(data) {
    return
this.http.patch(`${this.apiUrl}/v1/customer/editCustomerAddre
ss`, data);
}
```

- **Usage**:

```
const updatedAddress = {
    addressId: "123",
    addressLine1: "456 New St",
    addressLine2: "Apt 7C",
    city: "Gotham",
    state: "NY",
    postalCode: "67890",
    country: "USA",
};

customerService
    .editCustomerAddress(updatedAddress)
    .subscribe((response) => {
```

```
        console.log("Address updated successfully:",
response);
    });
```

1.8.2.4.4. Send Email

- **Endpoint**: `/api/v1/customer/contactUs`
- **Method**: POST
- **Description**: Sends an email message from the customer to the customer service team.
- **Parameters**:
    - `data`: An object containing the email's content (e.g., message, subject).
        - Example structure:

```
{
    "subject": "Inquiry about Order #1234",
    "message": "I have a question regarding my recent order. Can
you help?"
}
```

- **Response**: Returns a success message if the email is sent successfully.
- **Authorization**: Customer authentication required.
- **Error Handling**: Returns an error message if the email fails to send (e.g., server error).
- **Sample Code**:

```
sendEmail(data) {
    return
this.http.post(`${this.apiUrl}/v1/customer/contactUs`, data);
}
```

- **Usage**:

```
const emailData = {
    subject: "Inquiry about Order #1234",
    message: "I have a question regarding my recent order.
Can you help?",
};
```

```
customerService.sendEmail(emailData).subscribe((response) =>
{
    console.log("Email sent successfully:", response);
});
```

1.8.2.4.5. Delete Address of Requesting Customer by ID

- **Endpoint**: `/api/v1/customer/deleteAddressOfRequestCustomerById/{id}`
- **Method**: DELETE
- **Description**: Deletes a specific address of the currently logged-in customer by its ID.
- **Parameters**:
    - `id`: The unique identifier of the address to be deleted.
- **Response**: Returns a confirmation message if the address is deleted successfully.
- **Authorization**: Customer authentication required.
- **Error Handling**: Returns an error message if the address cannot be deleted (e.g., invalid ID, address not found).
- **Sample Code**:

```
deleteAddressOfRequestCustomerById(id) {
    return
this.http.delete(`${this.apiUrl}/v1/customer/deleteAddressOfR
equestCustomerById/${id}`);
}
```

- **Usage**:

```
const addressId = "123";
customerService
    .deleteAddressOfRequestCustomerById(addressId)
    .subscribe((response) => {
        console.log("Address deleted successfully:",
response);
    });
```

1.8.2.4.6. Get Nearby Restaurants

- **Endpoint**: `/api/v1/customer/getNearbyRestaurants`
- **Method**: GET

- **Description**: Retrieves a list of restaurants near a specified latitude and longitude.
- **Parameters**:
    - `latitude`: The latitude of the customer's location.
    - `longitude`: The longitude of the customer's location.
- **Response**: Returns an array of restaurant details located near the specified location.
- **Authorization**: Customer authentication required.
- **Error Handling**: Returns an error message if no restaurants are found or if the request fails.
- **Sample Code**:

```
getNearbyRestaurants(latitude, longitude) {
    return
this.http.get(`${this.apiUrl}/v1/customer/getNearbyRestaurant
s?latitude=${latitude}&longitude=${longitude}`);
}
```

- **Usage**:

```
const latitude = 40.7128;
const longitude = -74.006;

customerService
    .getNearbyRestaurants(latitude, longitude)
    .subscribe((restaurants) => {
        console.log("Nearby restaurants:", restaurants);
    });
```

1.8.2.4.7. Get All Restaurants

- **Endpoint**: `/api/v1/customer/getAllRestaurants`
- **Method**: GET
- **Description**: Retrieves a list of all available restaurants.
- **Parameters**: None.
- **Response**: Returns an array of restaurant details.
- **Authorization**: Customer authentication required.
- **Error Handling**: Returns an error message if the request fails.
- **Sample Code**:

```
getAllRestaurants() {
    return
this.http.get(`${this.apiUrl}/v1/customer/getAllRestaurants`)
;
}
```

- **Usage**:

```
customerService.getAllRestaurants().subscribe((restaurants)
=> {
    console.log("All restaurants:", restaurants);
});
```

1.8.2.4.8. Get Restaurant Details by URL

- **Endpoint**: `/api/v1/customer/getRestaurantDetailsFromRestaurantUrl/{restaurantUrl}`
- **Method**: GET
- **Description**: Retrieves the details of a specific restaurant using its URL.
- **Parameters**:
  - `restaurantUrl`: The unique URL of the restaurant.
- **Response**: Returns the details of the restaurant.
- **Authorization**: Customer authentication required.
- **Error Handling**: Returns an error message if the restaurant cannot be found.
- **Sample Code**:

```
getRestaurantDetailsFromRestaurantUrl(restaurantUrl) {
    return
this.http.get(`${this.apiUrl}/v1/customer/getRestaurantDetail
sFromRestaurantUrl/${restaurantUrl}`);
}
```

- **Usage**:

```
const restaurantUrl = "some-restaurant-url";

customerService
```

```
    .getRestaurantDetailsFromRestaurantUrl(restaurantUrl)
    .subscribe((details) => {
        console.log("Restaurant details:", details);
    });
```

1.8.2.4.9. Get Restaurant Details by ID

- **Endpoint**: `/api/v1/customer/getRestaurantDetailsFromRestaurantId/{restaurantId}`
- **Method**: GET
- **Description**: Retrieves the details of a specific restaurant using its ID.
- **Parameters**:
  - `restaurantId`: The unique ID of the restaurant.
- **Response**: Returns the details of the restaurant.
- **Authorization**: Customer authentication required.
- **Error Handling**: Returns an error message if the restaurant cannot be found.
- **Sample Code**:

```
getRestaurantDetailsFromRestaurantId(restaurantId) {
    return
this.http.get(`${this.apiUrl}/v1/customer/getRestaurantDetail
sFromRestaurantId/${restaurantId}`);
}
```

- **Usage**:

```
const restaurantId = "12345";

customerService
    .getRestaurantDetailsFromRestaurantId(restaurantId)
    .subscribe((details) => {
        console.log("Restaurant details:", details);
    });
```

1.8.2.4.10. Get Promo Codes for Restaurant by URL

- **Endpoint**: `/api/v1/customer/getPromoCodesForRestaurantUrl/{restaurantUrl}`
- **Method**: GET

- **Description**: Retrieves a list of active promo codes for a specific restaurant using its URL.
- **Parameters**:
  - `restaurantUrl`: The unique URL of the restaurant.
- **Response**: Returns an array of promo code details.
- **Authorization**: Customer authentication required.
- **Error Handling**: Returns an error message if no promo codes are available or if the request fails.
- **Sample Code**:

```
getPromoCodesForRestaurantUrl(restaurantUrl) {
    return
this.http.get(`${this.apiUrl}/v1/customer/getPromoCodesForRes
taurantUrl/${restaurantUrl}`);
}
```

- **Usage**:

```
const restaurantUrl = "some-restaurant-url";

customerService
    .getPromoCodesForRestaurantUrl(restaurantUrl)
    .subscribe((promoCodes) => {
        console.log("Promo codes:", promoCodes);
    });
```

1.8.2.4.11. Check If Promo Code is Valid

- **Endpoint**: `/api/v1/customer/checkIfPromoCodeIsValid`
- **Method**: POST
- **Description**: Validates a promo code for a specific restaurant and order amount.
- **Parameters**:
  - `data`: An object containing the promo code, order amount, and restaurant URL.
    - Example structure: `json { "promoCodeName": "SAVE20", "amountToBePaid": 100, "restaurantUrl": "some-restaurant-url" }`

- **Response**: Returns a success message with promo code validity details or an error message if invalid.
- **Authorization**: Customer authentication required.
- **Error Handling**: Returns an error message if the promo code is invalid or the request fails.
- **Sample Code**:

```
checkIfPromoCodeIsValid(promoCodeName, amountToBePaid,
restaurantUrl) {
    const data = { promoCodeName, amountToBePaid,
restaurantUrl };
    return
this.http.post(`${this.apiUrl}/v1/customer/checkIfPromoCodeIs
Valid`, data);
}
```

- **Usage**:

```
const promoCodeData = {
    promoCodeName: "SAVE20",
    amountToBePaid: 100,
    restaurantUrl: "some-restaurant-url",
};

customerService
    .checkIfPromoCodeIsValid(
        promoCodeData.promoCodeName,
        promoCodeData.amountToBePaid,
        promoCodeData.restaurantUrl
    )
    .subscribe((response) => {
        console.log("Promo code validity:", response);
    });
```

1.8.2.4.12. Update Customer Data

- **Endpoint**: `/api/v1/customer/updateCustomerData`
- **Method**: POST
- **Description**: Updates the personal information of the currently logged-in customer.
- **Parameters**:
    - `data`: An object containing the updated customer information.

        - Example structure: `json { "name": "John Doe", "email": "john.doe@example.com", "phone": "1234567890" }`

- **Response**: Returns a success message with the updated customer details.
- **Authorization**: Customer authentication required.
- **Error Handling**: Returns an error message if the update fails (e.g., invalid data or server error).
- **Sample Code**:

```
updateCustomerData(data) {
    return
this.http.post(`${this.apiUrl}/v1/customer/updateCustomerData
`, data);
}
```

- **Usage**:

```
const customerData = {
    name: "John Doe",
    email: "john.doe@example.com",
    phone: "1234567890",
};

customerService.updateCustomerData(customerData).subscribe((r
esponse) => {
    console.log("Customer data updated successfully:",
response);
});
```

1.8.2.4.13. Check If Dine-In is Available

- **Endpoint**: `/api/v1/customer/isDineInAvailable/{restaurantId}`
- **Method**: GET
- **Description**: Checks if dine-in service is available at a specific restaurant.
- **Parameters**:
    - `restaurantId`: The unique ID of the restaurant.
- **Response**: Returns a boolean value indicating whether dine-in is available or not.
- **Authorization**: Customer authentication required.

- **Error Handling**: Returns an error message if the request fails (e.g., invalid restaurant ID or server error).
- **Sample Code**:

```
isDineInAvailable(restaurantId) {
    return
this.http.get(`${this.apiUrl}/v1/customer/isDineInAvailable/$
{restaurantId}`);
}
```

- **Usage**:

```
const restaurantId = "12345";

customerService.isDineInAvailable(restaurantId).subscribe((is
Available) => {
    console.log(`Dine-in availability: ${isAvailable}`);
});
```

1.8.2.4.14. Get Restaurant Status

- **Endpoint**: `/api/v1/customer/getRestaurantStatus/{restaurantId}`
- **Method**: GET
- **Description**: Retrieves the current status of a restaurant (e.g., open or closed).
- **Parameters**:
    - `restaurantId`: The unique ID of the restaurant.
- **Response**: Returns an object containing the restaurant's current status and other relevant information.
- **Authorization**: Customer authentication required.
- **Error Handling**: Returns an error message if the request fails (e.g., invalid restaurant ID or server error).
- **Sample Code**:

```
getRestaurantStatus(restaurantId) {
    return
this.http.get(`${this.apiUrl}/v1/customer/getRestaurantStatus
/${restaurantId}`);
}
```

- **Usage**:

```
const restaurantId = "12345";

customerService.getRestaurantStatus(restaurantId).subscribe((
status) => {
    console.log("Restaurant status:", status);
});
```

1.8.2.5. Google Maps Service Endpoints

1.8.2.5.1. Get Autocomplete Results

- **Endpoint**: `/api/v1/google-maps/autocomplete`
- **Method**: GET
- **Description**: Retrieves search suggestions based on the user's input.
- **Parameters**:
    - `input`: The user's search query.
    - **Response**: Returns an array of prediction objects.
    - **Authorization**: No authentication required.
    - **Error Handling**: Returns an empty array if no suggestions are found or if the request fails.
    - **Sample Code**:

```
getAutocompleteResults(query: string) {
    if (query) {
        const autocompleteUrl =
`${this.apiUrl}/v1/google-maps/autocomplete?
input=${query}`;
        return this.httpClient.get<any>
(autocompleteUrl).pipe(
            map((response) => response.predictions),
            catchError(() => of([]))
        );
    } else {
        return of([]);
    }
}
```

- **Usage**:

```
googleMapsService.getAutocompleteResults("New
York").subscribe((results) => {
    console.log("Autocomplete results:", results);
});
```

1.8.2.5.2. Get Geocode Details

- **Endpoint**: `/api/v1/google-maps/geocode-details`
- **Method**: GET
- **Description**: Retrieves the geocode details (address components) for a specific latitude and longitude.
- **Parameters**:
  - `latitude`: The latitude of the location.
  - `longitude`: The longitude of the location.
- **Response**: Returns an object with the geocode details.
- **Authorization**: No authentication required.
- **Error Handling**: Returns an empty object if the details cannot be retrieved or if the request fails.
- **Sample Code**:

```
getGeocodeDetails(latitude: number, longitude: number) {
    let url = `${this.apiUrl}/v1/google-maps/geocode-details?
latitude=${latitude}&longitude=${longitude}`;

    return this.httpClient.get<any>(url).pipe(
        map((response) => {
            console.log("Geocoding API response:", response);
            return response;
        }),
        catchError((error) => {
            console.error("Geocoding API error:", error);
            return of("");
        })
    );
}
```

- **Usage**:

```
const latitude = 40.7128;
const longitude = -74.006;
```

```
googleMapsService.getGeocodeDetails(latitude,
longitude).subscribe((details) => {
    console.log("Geocode details:", details);
});
```

1.8.2.5.3. Get Formatted Geocode Details

- **Endpoint**: `/api/v1/google-maps/geocode-details`
- **Method**: GET
- **Description**: Retrieves the formatted geocode details (address components) for a specific latitude and longitude.
- **Parameters**:
    - `latitude`: The latitude of the location.
    - `longitude`: The longitude of the location.
    - **Response**: Returns an object with the formatted geocode details.
    - **Authorization**: No authentication required.
    - **Error Handling**: Returns an empty object if the details cannot be retrieved or if the request fails.
    - **Sample Code**:

```
getFormattedGeocodeDetails(latitude: number, longitude:
number) {
    let url = `${this.apiUrl}/v1/google-maps/geocode-
details?latitude=${latitude}&longitude=${longitude}`;

    return this.httpClient.get<any>(url).pipe(
        map((response) => {
            console.log("Geocoding API response:",
response);
            const addressComponents =
response.results[0].address_components;

            const postalCodeComponent =
addressComponents.find(
                (component) =>
component.types.includes("postal_code")
            );

            return {
                pinCode: postalCodeComponent ?
postalCodeComponent.long_name : "",
                completeAddress:
```

```
                response.results[0].formatted_address,
                    city: addressComponents.find((component)
=>
                        component.types.includes("locality")
                    )
                        ? addressComponents.find((component)
=>

component.types.includes("locality")
                          ).long_name
                        : "",
                    state:
addressComponents.find((component) =>

component.types.includes("administrative_area_level_1")
                    )
                        ? addressComponents.find((component)
=>

component.types.includes("administrative_area_level_1")
                          ).long_name
                        : "",
                    country:
addressComponents.find((component) =>
                        component.types.includes("country")
                    )
                        ? addressComponents.find((component)
=>

component.types.includes("country")
                          ).long_name
                        : "",
                };
            }),
            catchError((error) => {
                console.error("Geocoding API error:",
error);
                return of("");
            })
        );
    }
```

- **Usage**:

```
const latitude = 40.7128;
const longitude = -74.006;

googleMapsService.getFormattedGeocodeDetails(latitude,
longitude).subscribe((details) => {
    console.log("Formatted geocode details:", details);
});
```

1.8.2.5.4. Get Place Details

- **Endpoint**: `/api/v1/google-maps/place-details`
- **Method**: GET
- **Description**: Retrieves detailed information about a place using its place ID.
- **Parameters**:
  - `placeId`: The unique ID of the place.
  - **Response**: Returns an object with the place details.
  - **Authorization**: No authentication required.
  - **Error Handling**: Returns an empty object if the details cannot be retrieved or if the request fails.
  - **Sample Code**:

```
getPlaceDetails(placeId: string) {
    const placeDetailsUrl = `${this.apiUrl}/v1/google-
maps/place-details?placeId=${placeId}`;
    return this.httpClient.get<any>
(placeDetailsUrl).pipe(
        map((response) => response.result),
        catchError(() => of([]))
    );
}
```

- **Usage**:

```
const placeId = "some-place-id";

googleMapsService.getPlaceDetails(placeId).subscribe((details
) => {
    console.log("Place details:", details);
});
```

1.8.3. Error Codes and Handling

1.8.4. How to Test APIs as a Beginner

1.9. Database Design

1.9.1. Database Schema Overview

1.9.2. Key Tables and Their Purpose

1.9.3. Entity-Relationship Diagrams (ERD)

1.9.4. Sample Queries for Common Use Cases

1.10. User Interface (UI)

1.10.1. Screenshots of All Pages (annotated with descriptions)

1. **Login Page**
    - User authentication interface
    - Phone number input
    - OTP verification
2. **Menu Page**
    - Category-wise menu items
    - Item details with images
    - Add to cart functionality
3. **Cart Page**
    - Order summary
    - Item quantity adjustment
    - Checkout process
4. **Admin Dashboard**
    - Order management
    - Menu management
    - Analytics overview

1.10.2. Navigation Map

1.10.3. Design Principles Used

1. **Material Design**
    - Consistent UI components
    - Responsive layouts
    - Intuitive interactions
2. **User Experience**
    - Clear navigation
    - Fast loading
    - Error handling

1.11. Ad Hoc Process Configuration

1.11.1. Payment Gateway Integration

1.11.1.1. Overview of Payment Gateway Used

- Razorpay integration
- Secure payment processing
- Multiple payment methods

1.11.1.2. API Keys, Credentials, and Configuration Steps

1. Obtain Razorpay API keys
2. Configure in environment files
3. Set up webhook endpoints

1.11.1.3. Step-by-Step Guide for Setting Up Payment Flow

1. Initialize Razorpay
2. Create order
3. Handle payment response
4. Verify payment status

1.11.2. Messaging Service Integration (e.g., SMS, WhatsApp)

1.11.2.1. Overview of Messaging Providers

- WhatsApp Business API
- Firebase Cloud Messaging
- SMS gateway integration

1.11.2.2. Setting Up API Access and Authentication

1. WhatsApp Business account setup
2. API key configuration
3. Template message approval

1.11.2.3. Sending Messages

```
async function sendWhatsAppMessage(
    to: string,
    template: string,
    params: any[]
) {}
```

1.12. Testing Guidelines

1.12.1. Overview of Testing Strategy

1. **Unit Testing**

- Component testing
- Service testing
- Utility function testing

2. **Integration Testing**

- API endpoint testing
- Database operations
- Authentication flow

1.12.2. Functional Testing Scenarios

1. **Order Flow Testing**

- Menu item selection
- Cart operations
- Checkout process
- Payment integration

2. **Admin Operations**

- Menu management
- Order processing
- User management

1.12.3. Technical Testing

1. **Performance Testing**

- Load time optimization
- API response times
- Database query performance

2. **Security Testing**

- Authentication
- Authorization
- Data encryption

1.12.4. Bug Reporting Guidelines

1. **Bug Report Format**

- Title: Short description of the issue
- Description: Detailed explanation of the problem
- Steps to Reproduce: Step-by-step guide to reproduce the bug
- Expected Behavior: What should happen
- Actual Behavior: What is happening
- Screenshots: Visual evidence of the bug
- Environment: Browser, device, OS
- Severity: Low/Medium/High

- Priority: Low/Medium/High

1.13. Deployment and Maintenance

1.13.1. Deployment Process

1. **Build Process**

```
ng build --configuration production
```

2. **Firebase Deployment**

```
firebase use production
firebase deploy
```

1.13.2. Version Control Guidelines

1. **Branch Strategy**
   - main: production
   - develop: development
   - feature branches: new features
   - bugfix branches: bug fixes
   - hotfix branches: critical fixes
   - release branches: version releases
2. **Commit Messages**
   - feat: new feature
   - fix: bug fix
   - docs: documentation
   - style: formatting
   - refactor: code restructuring

1.13.3. Backup and Recovery Plan

1. **Database Backup**

   - Daily automated backups
   - Manual backup before major updates
   - Backup verification process

2. **Recovery Procedures**

   - Database restoration
   - Application rollback
   - Emergency contacts

1.14. Troubleshooting Guide

1.14.1. Common Issues and Fixes

1. **Authentication Issues**
    - Check Firebase configuration
    - Verify API keys
    - Clear browser cache
2. **Payment Issues**
    - Verify Razorpay integration
    - Check webhook configuration
    - Monitor payment logs

1.14.2. Debugging Tips for Developers

1. **Frontend Debugging**
    - Use Chrome DevTools
    - Check console logs
    - Monitor network requests
2. **Backend Debugging**
    - Firebase Functions logs
    - Database queries
    - API responses

1.15. Security Considerations

1.15.1. Security Practices Implemented

1. **Authentication**
    - Phone number verification
    - JWT token management
    - Session handling
2. **Data Security**
    - HTTPS encryption
    - Firebase security rules
    - Input validation

1.15.2. Guidelines for Handling Sensitive Data

1. **User Data**
    - Encryption at rest
    - Secure transmission
    - Access control

2. **Payment Information**
    - PCI compliance
    - Tokenization
    - Secure storage

1.16. FAQ

1.16.1. Common Questions by Non-Technical Staff

Q: How do I update menu items? A: Use the admin dashboard menu management section.

Q: How do I process orders? A: Monitor the order management dashboard and update order statuses.

1.16.2. Questions Related to API Usage

Q: How do I test API endpoints? A: Use Postman or the Firebase Emulator Suite.

Q: How do I handle API errors? A: Check error codes and implement proper error handling.

1.16.3. Testing and Debugging FAQs

Q: How do I run tests? A: Use `ng test` for unit tests.

Q: How do I debug issues? A: Use browser DevTools and Firebase Console.

1.17. Appendix

1.17.1. Resources and References

1. **Documentation**
    - Angular Documentation
    - Firebase Documentation
    - Razorpay Documentation
2. **Tutorials**
    - Angular tutorials
    - Firebase guides
    - Testing guides

1.17.2. Links to Tools, Libraries, and Frameworks Used

1. **Development Tools**
    - Visual Studio Code
    - Git
    - Node.js
2. **Frameworks and Libraries**

    - Angular
    - Angular Material
    - Firebase

1.17.3. Glossary of Technical Terms

- **Angular**: Frontend framework
- **Firebase**: Backend platform
- **API**: Application Programming Interface
- **JWT**: JSON Web Token
- **REST**: Representational State Transfer
- **OTP**: One-Time Password
- **UI/UX**: User Interface/User Experience
- **CI/CD**: Continuous Integration/Continuous Deployment