**Solution for Coffee Shop Chain**

1. **Use Case**

**Use Case 1: Shop Owner Setup**

**Actor**: Shop Owner

**Description**: The shop owner sets up and configures the app for their specific shop.

**Main Flow**:

1. Shop owner logs into the Shop Owner App.
2. Shop owner navigates to the setup/configuration section.
3. Shop owner enters location and contact details.
4. Shop owner customizes the coffee menu and pricing.
5. Shop owner specifies the number of queues and maximum queue size.
6. Shop owner sets opening and closing times.

**Use Case 2: Shop Operator Queue Management**

**Actor**: Shop Operator

**Description**: The shop operator manages the queue and monitors waiting customers.

**Main Flow**:

1. Shop operator logs into the Shop Owner App.
2. Shop operator accesses the queue management section.
3. Shop operator views the current queue size and number of waiting customers.
4. Shop operator views the orders placed by customers in the queue.
5. Shop operator removes customers from the queue for service.

**Use Case 3: Customer Registration**

**Actor**: Customer

**Description**: The customer registers with the Customer App.

**Main Flow**:

1. Customer opens the Customer App.
2. Customer selects the registration option.
3. Customer enters their mobile number, name, and regular address.
4. Customer completes the registration process.

**Use Case 4: Customer Order Placement**

**Actor**: Customer

**Description**: The customer places an order using the Customer App.

**Main Flow**:

1. Customer logs into the Customer App.
2. Customer locates nearby coffee shops.
3. Customer selects a coffee shop and views the menu.
4. Customer selects items from the menu and places an order.
5. Customer monitors their position in the queue and expected wait time.
6. Customer has the option to cancel the order and exit the queue.

**Use Case 5: Customer Queue Monitoring**

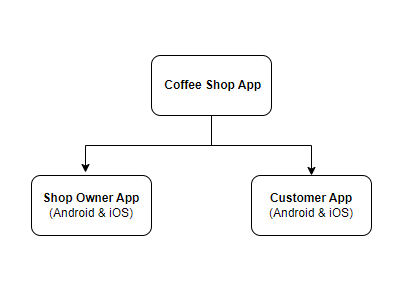
**Actor**: Customer

**Description**: The customer monitors their position in the queue and expected wait time.

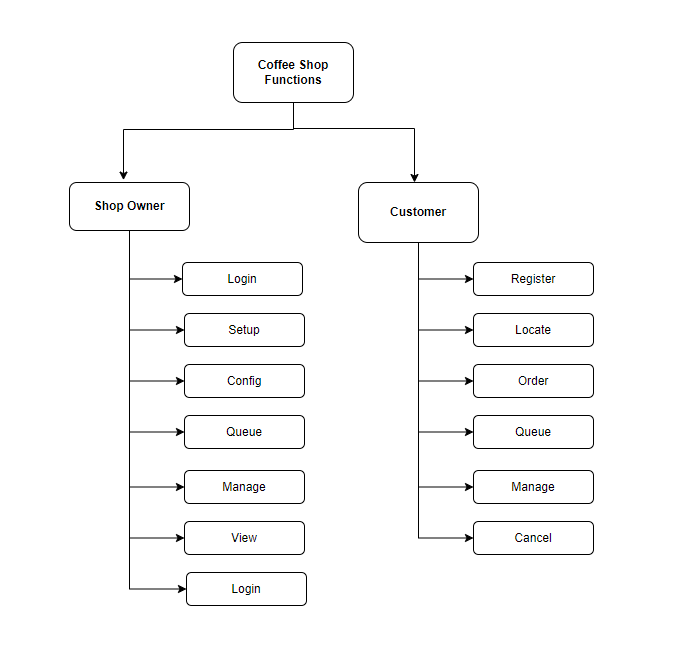
**Main Flow**:

1. Customer logs into the Customer App.
2. Customer selects the option to view their position in the queue.
3. Customer views their current position and expected wait time.
4. Customer can decide to cancel the order and exit the queue if desired.
5. **Concept Diagram**

### **Concept Diagram 1: Coffee Shop App Architecture**

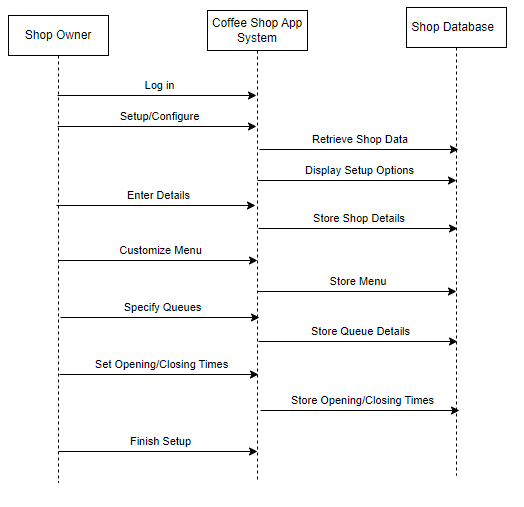


### **Concept Diagram 2: Coffee Shop App Functionality**

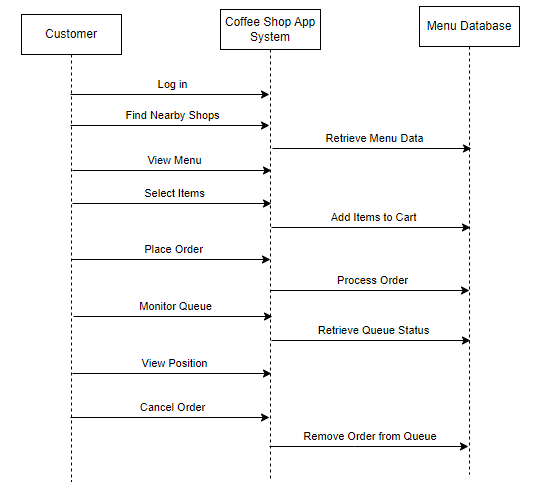


1. **Sequence Diagram**

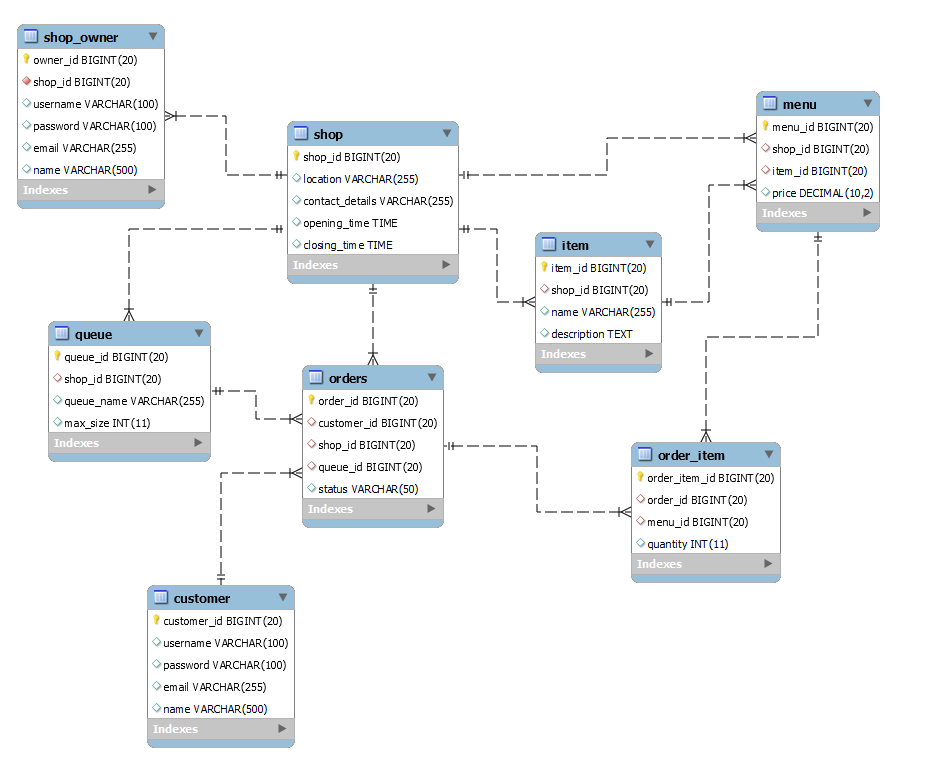
### **Sequence Diagram 1: Shop Owner Setup**

****

### **Sequence Diagram 2: Customer Order Placement**

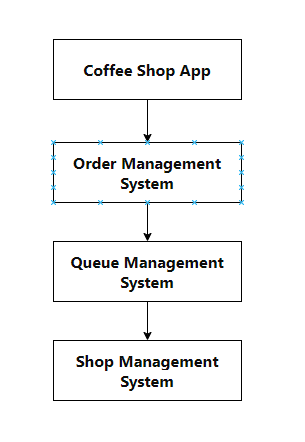
****

1. **Data designs**

****

1. **Data flows**

**Coffee Shop App DFD**

****

**In this diagram:**

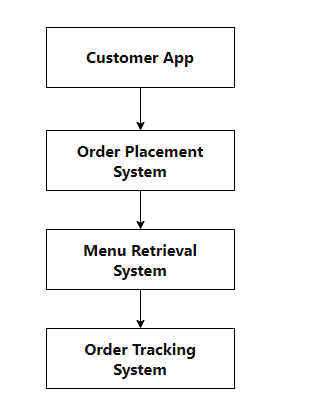
The Coffee Shop App interacts with three main systems: Order Management, Queue Management, and Shop Management.

The Coffee Shop App sends orders to the Order Management System, which then manages the processing of orders.

The Order Management System communicates with the Queue Management System to manage the queue of orders.

The Queue Management System, in turn, communicates with the Shop Management System for various shop-related tasks.

**Customer App DFD:**

****

**In this diagram:**

The Customer App interacts with three main systems: Order Placement, Menu Retrieval, and Order Tracking.

The Order Placement System allows customers to place orders.

The Menu Retrieval System provides customers with access to the coffee shop's menu.

The Order Tracking System allows customers to track the status of their orders.

These DFDs provide a simplified overview of the interactions between the apps and the systems within the coffee shop environment.

1. **API endpoints**
2. **Authentication Endpoints**
   * **POST /api/auth/login**: Authenticate and log in a customer.
   * **POST /api/auth/logout**: Log out the currently authenticated customer.
   * **POST /api/auth/register**: Register a new customer account.
   * **POST /api/auth/shop-owner/login**: Authenticate and log in a shop owner.
   * **POST /api/auth/ shop-owner/logout**: Log out the currently authenticated a shop owner.
   * **POST /api/auth/ shop-owner/register**: Register a new a shop owner account.
3. **Shop Endpoints**:
   * **GET /api/shops**: Retrieve a list of all coffee shops.
   * **GET /api/shops/{shop\_id}**: Retrieve details of a specific coffee shop by ID.
   * **POST /api/shops**: Create a new coffee shop.
   * **PUT /api/shops/{shop\_id}**: Update details of an existing coffee shop.
   * **DELETE /api/shops/{shop\_id}**: Delete a coffee shop by ID.
4. **Shop Owner Endpoints**
   * **GET /api/shop-owners**: Retrieve a list of all shop owners.
   * **GET /api/shop-owners/{owner\_id}**: Retrieve details of a specific shop owner by ID.
   * **POST /api/shop-owners**: Create a new shop owner.
   * **PUT /api/shop-owners/{owner\_id}**: Update details of an existing shop owner.
   * **DELETE /api/shop-owners/{owner\_id}**: Delete a shop owner by ID.
5. **Queue Endpoints**:
   * **GET /api/queues**: Retrieve a list of all queues across coffee shops.
   * **GET /api/queues/{queue\_id}**: Retrieve details of a specific queue by ID.
   * **POST /api/queues**: Create a new queue for a coffee shop.
   * **PUT /api/queues/{queue\_id}**: Update details of an existing queue.
   * **DELETE /api/queues/{queue\_id}**: Delete a queue by ID.
   * **GET /api/customers/{customer\_id}/queues**: Retrieve information about queues the customer is in.
   * **POST /api/customers/{customer\_id}/join\_queue/{queue\_id}**: Join a queue for a coffee shop.
   * **POST /api/customers/{customer\_id}/leave\_queue/{queue\_id}**: Leave a queue for a coffee shop.
6. **Customer Endpoints**:
   * **GET /api/customers**: Retrieve a list of all customers.
   * **GET /api/customers/{customer\_id}**: Retrieve details of a specific customer by ID.
   * **POST /api/customers**: Create a new customer.
   * **PUT /api/customers/{customer\_id}**: Update details of an existing customer.
   * **DELETE /api/customers/{customer\_id}**: Delete a customer by ID.
7. **Item Endpoints**:
   * **GET /api/items**: Retrieve a list of all items available in the coffee shops.
   * **GET /api/items/{item\_id}**: Retrieve details of a specific item by ID.
   * **POST /api/items**: Create a new item for a coffee shop's menu.
   * **PUT /api/items/{item\_id}**: Update details of an existing item.
   * **DELETE /api/items/{item\_id}**: Delete an item by ID.
8. **Menu Endpoints**:
   * **GET /api/menus**: Retrieve menus of all coffee shops.
   * **GET /api/menus/{shop\_id}**: Retrieve menu of a specific coffee shop by ID.
   * **POST /api/menus/{shop\_id}**: Create a new menu for a coffee shop.
   * **PUT /api/menus/{shop\_id}**: Update menu of a specific coffee shop.
   * **DELETE /api/menus/{shop\_id}**: Delete menu of a specific coffee shop.
9. **Order Endpoints**:
   * **GET /api/orders**: Retrieve a list of all orders.
   * **GET /api/orders/{order\_id}**: Retrieve details of a specific order by ID.
   * **POST /api/orders**: Place a new order.
   * **PUT /api/orders/{order\_id}**: Update details of an existing order (e.g., change status).
   * **DELETE /api/orders/{order\_id}**: Cancel an order by ID.
   * **GET /api/customers/{customer\_id}/orders**: Retrieve a list of orders placed by a specific customer.
   * **POST /api/customers/{customer\_id}/orders**: Place a new order for the customer.
10. **Order Item Endpoints**:
    * **GET /api/orders/{order\_id}/items**: Retrieve a list of items included in a specific order.
    * **POST /api/orders/{order\_id}/items**: Add a new item to an existing order.
    * **PUT /api/orders/{order\_id}/items/{item\_id}**: Update details of an item in the order.
    * **DELETE /api/orders/{order\_id}/items/{item\_id}**: Remove an item from the order.
11. **Order Processing Endpoints**:
    * **POST /api/order-processing/process-orders**: Get Order From Queue to process
12. **Time estimation**

|  |  |  |
| --- | --- | --- |
| **Tasks** | **Sub-tasks** | **Days** |
| **Requirement Analysis and Planning** |  |  |
|  | Reviewing the solution requirements and preparing a detailed plan | 2 |
| **Database Design** |  |  |
|  | Designing the database schema and relationships | 1 |
|  | Writing DDL scripts and setting up the database | 1 |
| **Backend Development** |  |  |
|  | Setting up the Spring Boot project structure | 1 |
|  | Implementing Authentication Endpoints | 3 |
|  | Implementing Shop Endpoints | 2 |
|  | Implementing Shop Owner Endpoints | 2 |
|  | Implementing Queue Endpoints | 2 |
|  | Implementing Customer Endpoints | 2 |
|  | Implementing Item Endpoints | 2 |
|  | Implementing Menu Endpoints | 2 |
|  | Implementing Order Endpoints | 2 |
|  | Implementing Order Item Endpoints | 2 |
|  | Implementing Order Processing Endpoints | 2 |
|  |  |  |
| **Testing** |  |  |
|  | Writing unit tests for backend components | 2 |
|  |
|  | Performing integration tests and API testing | 2 |
|  | Testing data consistency, performance, security, and error handling | 2 |
| **Documentation** |  |  |
|  | Documenting the API endpoints and usage | 1 |
|  | Writing README and setup instructions | 1 |
| **Deployment** |  |  |
|  | Setting up deployment environment (AWS, Docker) | 1 |
|  | Deploying the solution and configuring Docker containers | 1 |
| **Buffer for Contingencies** |  |  |
|  | Additional time for unforeseen issues, debugging, and refinements | 2 |
| **TOTAL** | | **38** |

1. **Stardard**
2. **Coding Standards**:
   * Follow a consistent coding style and formatting convention (e.g., Google Java Style Guide, Spring Framework conventions).
   * Use meaningful and descriptive names for variables, methods, classes, and packages.
   * Write modular and reusable code with clear separation of concerns.
   * Avoid using hardcoded values and magic numbers; use constants or configuration properties instead.
   * Ensure proper error handling and logging throughout the application.
3. **Naming Conventions:**
   * Use camelCase for variable and method names.
   * Use PascalCase for class and interface names.
   * Use lowercase letters for package names.
   * Choose descriptive and meaningful names that reflect the purpose and functionality of the components.
4. **Security Standards:**
   * Implement authentication and authorization mechanisms to secure the APIs (e.g., JWT, OAuth 2.0).
   * Sanitize user input to prevent SQL injection, cross-site scripting (XSS), and other security vulnerabilities.
   * Use HTTPS for secure communication between the client and server.
   * Implement role-based access control (RBAC) to restrict access to sensitive resources.
   * Regularly update dependencies and libraries to patch known security vulnerabilities.
5. **Technology Standards:**
   * Use Spring Boot for building RESTful APIs, as it provides a robust framework with built-in support for dependency injection, MVC architecture, and security features.
   * Utilize Spring Data JPA for database access, which simplifies data access code and reduces boilerplate.
   * Choose PostgreSQL as the database, as it offers features like ACID compliance, data integrity, and scalability.
   * Containerize the application using Docker for easier deployment and scalability.
   * Deploy the application on AWS (Amazon Web Services) for cloud hosting, leveraging services like EC2, RDS, and ECS/EKS.
6. **Documentation Standards:**
   * Document the code using comments to explain complex logic, algorithms, or business rules.
   * Generate API documentation using tools like Swagger/OpenAPI to provide a clear description of the API endpoints, request/response formats, and authentication requirements.
   * Maintain a README file with instructions for setting up the development environment, running tests, and deploying the application.
7. **Security solution**
8. **Authentication and Authorization**:
   * Implement token-based authentication using JSON Web Tokens (JWT) or OAuth 2.0.
   * Users (shop owners, customers) will be required to authenticate themselves with valid credentials (e.g., username/password) to access protected endpoints.
   * Use role-based access control (RBAC) to enforce authorization rules and restrict access to specific resources based on user roles and permissions.
9. **HTTPS Encryption**:
   * Enforce HTTPS protocol for secure communication between clients and the server.
   * Utilize SSL/TLS certificates to encrypt data transmitted over the network, preventing eavesdropping and man-in-the-middle attacks.
10. **Input Validation and Sanitization**:
    * Validate and sanitize user input to prevent common security vulnerabilities such as SQL injection, cross-site scripting (XSS), and command injection.
    * Implement server-side validation for all incoming requests to ensure that only valid and expected data is processed.
11. **Password Hashing**:
    * Hash and salt passwords before storing them in the database to prevent unauthorized access in case of a data breach.
    * Use strong cryptographic hashing algorithms (e.g., bcrypt, Argon2) with a sufficient number of iterations to make brute-force attacks impractical.
12. **Session Management**:
    * Implement secure session management techniques to manage user sessions and prevent session hijacking or fixation.
    * Use industry best practices for session handling, such as using secure cookies, setting session timeouts, and employing CSRF (Cross-Site Request Forgery) protection mechanisms.
13. **API Security**:
    * Secure API endpoints by requiring authentication tokens for access.
    * Implement rate limiting and throttling to mitigate denial-of-service (DoS) attacks and prevent abuse of API resources.
    * Validate and sanitize API input parameters to prevent injection attacks and ensure data integrity.
14. **Security Headers**:
    * Set appropriate security headers in HTTP responses to enhance browser security and protect against common web vulnerabilities.
    * Headers such as Content Security Policy (CSP), X-Content-Type-Options, X-Frame-Options, and X-XSS-Protection can help prevent various types of attacks, such as XSS and clickjacking.
15. **Regular Security Audits and Updates**:
    * Conduct regular security audits and code reviews to identify and address potential security vulnerabilities.
    * Stay informed about the latest security threats and best practices, and apply security patches and updates promptly to mitigate known vulnerabilities.
16. **Testing**
17. **Unit Testing**:
    * Write unit tests for each component of the backend application, including controllers, services, and repositories.
    * Test individual functions and methods to ensure they return the expected results for various inputs.
    * Mock external dependencies such as databases and external API calls to isolate the unit tests.
18. **Integration Testing**:
    * Perform integration tests to verify that different components of the application work together correctly.
    * Test the interactions between controllers, services, and repositories to ensure that data flows correctly through the system.
    * Use an in-memory or embedded database for integration tests to simulate database interactions.
19. **API Testing**:
    * Test each API endpoint using tools like Postman, Insomnia, or automated testing frameworks.
    * Verify that the APIs accept valid input, handle errors appropriately, and return the expected responses.
    * Test various scenarios, including edge cases and error conditions, to ensure robustness.
20. **Data Consistency Testing**:
    * Ensure that data stored in the database is consistent and accurate.
    * Write tests to validate data integrity constraints such as foreign key constraints, unique constraints, and data validation rules.
21. **Performance Testing**:
    * Measure the performance of API endpoints under different load conditions.
    * Test the application's response time, throughput, and resource utilization to ensure scalability and responsiveness.
22. **Security Testing**:
    * Perform security testing to identify and address potential vulnerabilities in the application.
    * Test for common security issues such as SQL injection, cross-site scripting (XSS), and authentication/authorization bypass.
23. **Error Handling Testing**:
    * Test how the application handles errors and exceptions.
    * Verify that appropriate error messages are returned to the client and logged for debugging purposes.
24. **Deployment Testing**:
    * Test the deployment process to ensure that the application can be deployed successfully in different environments.
    * Verify that configuration settings are correct and that the application runs as expected after deployment.