**Customer service**

**Step 1: Design the User Entity**

1. **Purpose**:
   * The User entity will represent the credentials and roles of both customers and managers/admin.
   * It will store basic login details such as username, email (UNIQUE), and password.
2. **Fields to Include**:
   * id: A unique identifier for the user.
   * username: A unique identifier chosen by the user for login.
   * email: The user's email address.
   * password: An encrypted password for secure authentication.
   * roles: A collection of roles (ROLE\_CUSTOMER or ROLE\_MANAGER).
3. **Design Considerations**:
   * Ensure username and email are unique.
   * Use a collection to store multiple roles for each user.

**Step 2: Design the CustomerDetails Entity**

1. **Purpose**:
   * The CustomerDetails entity stores additional profile information for customers, separate from authentication data in the User entity.
2. **Fields to Include**:
   * id: A unique identifier for the profile details.
   * userId: A foreign key reference to the User entity.
   * firstName: The customer's first name.
   * lastName: The customer's last name.
   * phoneNumber: The customer's contact number.
   * address: The customer's address.
   * dob: The customer's date of birth (optional).
3. **Relationship**:
   * Establish a one-to-one relationship between User and CustomerDetails.
4. **Design Considerations**:
   * Keep CustomerDetails flexible to add more fields as needed.

**Step 3: Create Repositories**

1. **UserRepository**:
   * Handles database operations for the User entity.
   * Required Methods:
     + Find a user by username.
     + Check if username or email already exists.
     + Fetch users with specific roles.
2. **CustomerDetailsRepository**:
   * Handles database operations for the CustomerDetails entity.
   * Required Methods:
     + Fetch customer details by userId.

**Step 4: Define DTOs**

1. **UserRegistrationRequest**:
   * Captures details for registering a user.
   * Fields:
     + username, email, password, firstName, lastName, phoneNumber, address, and isManager.
2. **UserLoginRequest**:
   * Captures login credentials.
   * Fields:
     + username and password.
3. **UserResponse**:
   * Structures response data after registration or login.
   * Fields:
     + id, username, email, and roles.
4. **CustomerDetailsResponse**:
   * Captures profile details of a customer.
   * Fields:
     + firstName, lastName, phoneNumber, address, and dob.

**Step 5: Implement the Customer Service Logic**

1. **Responsibilities**:
   * Handle user registration by saving credentials and creating profile details.
   * Authenticate users during login.
   * Fetch profile details for a customer.
2. **User Registration Logic**:
   * Validate the uniqueness of username and email.
   * Encrypt the password before saving the user.
   * Assign roles (ROLE\_CUSTOMER or ROLE\_MANAGER) based on the isManager flag.
   * Create and save CustomerDetails for customers during registration.
3. **User Login Logic**:
   * Validate username and password during login.
   * Return user details (e.g., username, roles) for token generation at the API Gateway.
4. **Fetch Profile Logic**:
   * Retrieve CustomerDetails using the userId.
5. **Error Handling**:
   * Handle duplicate usernames, missing inputs, and invalid credentials gracefully.

**Step 6: Build REST API Endpoints**

1. **Endpoints**:
   * **POST /customers/register**:
     + Accept registration details and create both User and CustomerDetails.
   * **POST /customers/login**:
     + Authenticate the user and provide user details to the API Gateway for token generation.
   * **GET /customers/{userId}/details**:
     + Fetch the profile details of a customer using their userId.
2. **API Implementation Notes**:
   * Validate all inputs before processing.
   * Ensure meaningful responses for success and error cases.

**Step 7: Testing**

1. **Unit Testing**:
   * Test registration, login, and profile retrieval logic for both customers and managers.
   * Validate the behavior when duplicate usernames or emails are submitted.
2. **Integration Testing**:
   * Test the interaction between the Customer Service and the API Gateway for login and token generation.
   * Validate role-based access using tokens.