Complete Documentation

---

1. INTRODUCTION

The Barber Booking System is a comprehensive web application designed to facilitate barbershop management and customer appointment booking. The system leverages modern technologies including FastAPI for API development, SQLAlchemy for database management, and Pydantic for data validation.

This documentation provides a complete architectural overview of the system, detailing how each component works independently and in conjunction with other modules to deliver a seamless booking experience.

---

2. PROJECT STRUCTURE OVERVIEW

The project follows a well-organized directory structure that separates concerns and promotes maintainability:

Root Level Structure

The main application folder contains the following key directories and files:

- app: Main application directory containing all source code
- main.py: Application entry point
- Dockerfile: Container configuration for deployment
- requirements.txt: Python dependencies
- .env: Environment configuration variables
- .gitignore: Git ignore configuration
- tests: Directory containing test files

Application Internal Structure (app directory)

The application is organized into several logical layers:

- src: Source code containing all business logic and operations
- db: Database configuration and initialization files
- logs: Application logging directory
- pro: virtual environment
- tests: Unit and integration tests

Source Code Layer (src directory)

The src directory implements a two-layer architecture pattern:

- core: Core utilities and helpers
- db: Database models and base configurations
- routes: API endpoint definitions
- schemas: Request and response validation models
- services: Business logic layer
- utils: Utility functions and helpers

---

3. SYSTEM ARCHITECTURE

3.1 Architectural Pattern

The system employs a two-layer architecture pattern:

API Layer (Routes)

This layer handles HTTP requests and responses. Routes define all available endpoints and delegate business logic to the service layer.

Business Logic Layer (Services)

This layer contains the core business logic for each module. Services orchestrate operations, validate data, handle database transactions, and interact directly with the database using SQLAlchemy ORM sessions.

3.2 Data Flow Architecture

The request flow through the system follows this pattern:

1. Client sends HTTP request to API endpoint
2. Route handler receives and validates request using Pydantic schemas
3. Route delegates to appropriate service method with database session
4. Service applies business logic and executes database queries
5. Service returns processed data to route
6. Route formats response and returns HTTP response to client

---

4. MODULE DESCRIPTIONS

4.1 User Management Module

Purpose

Handles user registration, authentication, and email verification for both customers and shop owners.

Key Features

- User registration with email and phone validation
- Password-based authentication with secure hashing
- OTP-based email verification
- OTP-based login alternative
- User profile retrieval
- Role-based access control

Components

Routes File: user_routes.py Defines all API endpoints for user operations and connects them to service methods.

Service File: user_service.py Contains business logic for registration, authentication, OTP generation and verification.

Schema File: user_schema.py Defines request and response validation models using Pydantic.

4.2 Shop Management Module

Purpose

Enables shop owners to create and manage barbershops and allows customers to view available shops and book appointments.

Key Features

- Shop creation and management by owners
- Shop listing and retrieval
- Available slot management
- Slot booking with double booking prevention
- Shop operating hours management

Components

Routes File: shop_routes.py Defines all API endpoints for shop operations and booking.

Service File: shop_service.py Contains business logic for shop creation, retrieval, and slot management.

Schema File: shop_schemas.py Defines request and response validation models for shop operations.

4.3 Barber Management Module

Purpose

Manages barber information within shops including addition, updates, deletion, and availability management.

Key Features

- Add barbers to existing shops
- Update barber details and availability
- Delete barbers and associated records
- View available barbers
- Define barber working hours
- Set daily slot generation flag

Components

Routes File: barber_routes.py Defines all API endpoints for barber management operations.

Service File: barber_service.py Contains business logic for barber CRUD operations and availability management.

Schema File: barber_schemas.py Defines request and response validation models for barber operations.

4.4 Barber Slot Generation Module

Purpose

Automatically generates hourly appointment slots for barbers based on their defined working hours.

Key Features

- Automatic daily slot generation

- Hourly slot creation
- Prevention of duplicate slots
- Validation of shop operating status
- Selective single barber or batch generation
- Comprehensive error handling and logging

How It Works

Eligibility Criteria Barbers must meet the following conditions for slot generation:

- Daily generation flag is enabled
- Barber availability status is active
- Associated shop is open
- Barber working hours are fully defined

Generation Process

1. System checks for eligible barbers
2. For each eligible barber, validates shop status
3. Verifies barber working hours are complete
4. Generates one-hour slots from start time to end time
5. Skips any slots that fall in the past
6. Checks for existing slots to prevent duplication
7. Creates new slots with available status
8. Commits all changes to database

Slot Attributes Each generated slot contains:

- Barber ID and name
- Shop ID
- Date of slot
- Start time
- Slot status set to available
- Booking flag set to false

Error Handling

- Missing barbers: Logs informational message and exits
- Closed shops: Skips that barber with notification
- Incomplete hours: Logs warning and continues
- Database errors: Rolls back transaction and logs exception

Components

Routes File: slot_generator.py Handles slot generation triggering and management.

4.5 Booking Module

Purpose

Manages the complete booking process for customers to reserve barber appointment slots.

Key Features

- Multiple slot booking in single request
- Slot availability validation
- Double booking prevention
- Automatic booking record creation
- Complete audit trail with timestamps

Booking Process

Validation Phase For each requested slot, the system:

- Verifies slot exists in database
- Confirms slot belongs to specified barber
- Checks that slot is available and not previously booked
- Validates user, barber, and shop IDs

Booking Phase For each valid slot:

- Updates slot booking status to true
- Changes slot status to booked
- Creates booking record with user, barber, shop information
- Records booking timestamp in UTC format
- Stores booking confirmation status

Response Phase Returns confirmation including:

- Total number of slots successfully booked
- Details of each booked slot with ID, date, time, status
- Confirmation message

Data Consistency

All slot updates and booking record creations are performed as a single database transaction. If any slot booking fails, the entire transaction is rolled back.

Components

Routes File: booking_routes.py Defines booking API endpoints.

Service File: booking_service.py Contains business logic for slot booking and validation.

---

5. DATABASE LAYER

5.1 Database Models

User Model

Table Name: users

Fields:

- id: Integer, Primary Key, Auto-increment
- username: String(100), Unique, Indexed, Not Null
- email: String(150), Unique, Nullable
- hashed_password: String(255), Nullable
- phone_number: String(20), Unique, Nullable
- role: String(20), Default customer
- otp_code: String(10), Nullable
- otp_expiry: DateTime, Nullable
- is_verified: Boolean, Default False
- created_at: DateTime

Shop Model

Table Name: shops

Fields:

- shop_id: Integer, Primary Key, Auto-increment
- owner_id: Integer, Foreign Key to users.id, Not Null
- shop_name: String(200), Not Null
- address: Text, Not Null
- city: String(100), Not Null
- state: String(100), Not Null
- open_time: Time, Not Null
- close_time: Time, Not Null
- is_open: Boolean, Default True
- created_at: DateTime

Barber Model

Table Name: barbers

Fields:

- barber_id: Integer, Primary Key, Auto-increment
- barber_name: String(200), Not Null
- shop_id: Integer, Foreign Key to shops.shop_id, Cascade Delete, Not Null
- start_time: Time, Not Null
- end_time: Time, Not Null
- is_available: Boolean, Default True
- generate_daily: Boolean, Default False
- created_at: DateTime

BarberSlot Model

Table Name: barber_slots

Fields:

- slot_id: Integer, Primary Key, Auto-increment
- barber_id: Integer, Foreign Key to barbers.barber_id, Cascade Delete, Not Null
- shop_id: Integer, Foreign Key to shops.shop_id, Cascade Delete, Not Null
- slot_date: Date, Not Null
- slot_time: Time, Not Null
- is_booked: Boolean, Default False
- status: String(20), Default available
- created_at: DateTime

Booking Model

Table Name: bookings

Fields:

- booking_id: Integer, Primary Key, Auto-increment
- user_id: Integer, Foreign Key to users.id, Cascade Delete, Not Null
- barber_id: Integer, Foreign Key to barbers.barber_id, Cascade Delete, Not Null
- shop_id: Integer, Foreign Key to shops.shop_id, Cascade Delete, Not Null

- slot_id: Integer, Foreign Key to barber_slots.slot_id, Cascade Delete, Not Null
- booking_date: Date, Not Null
- booking_time: Time, Not Null
- status: String(20), Default booked
- created_at: DateTime

EmailVerification Model

Table Name: email_verification

Fields:

- id: Integer, Primary Key, Auto-increment
- email: String(150), Unique, Not Null
- otp_code: String(10), Not Null
- otp_expiry: DateTime, Not Null

BarberAvailability Model

Table Name: barber_availability

Fields:

- id: Integer, Primary Key, Auto-increment
- barber_id: Integer, Foreign Key to barbers.barber_id, Not Null
- available_date: Date, Not Null
- start_time: Time, Nullable
- end_time: Time, Nullable
- is_available: Boolean, Default True
- created_at: DateTime

Unique Constraint: barber_id and available_date

5.2 Database Configuration

Location: app/db directory

Key Files:

- init.py: Database session initialization and configuration
- base.py: SQLAlchemy declarative base and common model utilities
- database.py: Database connection setup and engine configuration

- models.py: All database model definitions
- test_database.py: Database testing utilities

Session Management Database sessions are created using SessionLocal for each request and automatically closed after request completion to ensure proper resource management.

---

6. COMPLETE API ENDPOINTS REFERENCE

6.1 USER MANAGEMENT ENDPOINTS

Register New User

Path: /register Method: POST Description: Registers a new user in the system with email and phone validation

Request Body: { "username": "John Doe", "email": "john@example.com", "password": "StrongPassword123", "phone_number": "9876543210", "role": "customer" }

Response (Success 200): { "msg": "User registered successfully" }

Response (Error 400): { "detail": "Email already registered" }

Send Email Verification OTP

Path: /send-verification-otp Method: POST Description: Generates and sends a six-digit OTP to user email

Request Body: { "email": "john@example.com", "role": "customer" }

Response (Success 200): { "msg": "Verification OTP sent" }

Response (Error 404): { "detail": "User not found" }

OTP Validity: Five minutes

Verify Email with OTP

Path: /verify-email Method: POST Description: Validates OTP and marks email as verified

Request Body: { "email": "john@example.com", "otp": "123456", "role": "customer" }

Response (Success 200): { "msg": "Email verified successfully" }

Response (Error 400): { "detail": "Invalid or expired OTP" }

Login with Password

Path: /login Method: POST Description: Authenticates user using email and password

Request Body: { "email": "john@example.com", "password": "StrongPassword123", "role": "customer" }

Response (Success 200): { "msg": "Login successful", "user_id": 1, "role": "customer" }

Response (Error 401): { "detail": "Invalid credentials" }

Request OTP for Login

Path: /otp-login Method: POST Description: Generates and sends OTP for passwordless login

Request Body: { "email": "john@example.com", "role": "customer" }

Response (Success 200): { "msg": "OTP sent successfully for customer" }

Response (Error 404): { "detail": "Email not found" }

Verify OTP for Login

Path: /verify-otp-login Method: POST Description: Validates OTP for login authentication

Request Body: { "email": "john@example.com", "otp": "654321", "role": "customer" }

Response (Success 200): { "msg": "Login successful as customer", "user_id": 1, "role": "customer" }

Response (Error 400): { "detail": "Invalid OTP" }

Get User Details

Path: /get_user Method: GET Description: Retrieves user profile information by email

Query Parameters: email: john@example.com

Response (Success 200): { "id": 1, "username": "John Doe", "email": "john@example.com", "phone_number": "9876543210" }

Response (Error 404): { "detail": "User not found" }

6.2 SHOP MANAGEMENT ENDPOINTS

Create New Shop

Path: /create Method: POST Description: Creates a new barbershop for the authenticated owner

Query Parameters: owner_id: 5

Request Body: { "shop_name": "Elite Barber Studio", "address": "MG Road", "city": "Bangalore", "state": "Karnataka", "open_time": "09:00:00", "close_time": "21:00:00" }

Response (Success 200): { "message": "Shop created successfully", "shop_id": 101 }

Response (Error 400): { "detail": "Owner already has a shop registered" }

Response (Error 404): { "detail": "Owner not found" }

Get All Shops

Path: /shops/ Method: GET Description: Retrieves list of all active shops in the system

Response (Success 200): [{ "shop_id": 101, "shop_name": "Elite Barber Studio", "address": "MG Road", "city": "Bangalore", "state": "Karnataka", "open_time": "09:00:00", "close_time": "21:00:00", "is_open": true }, { "shop_id": 102, "shop_name": "Urban Cuts", "address": "Jubilee Hills", "city": "Hyderabad", "state": "Telangana", "open_time": "10:00:00", "close_time": "22:00:00", "is_open": false }]

Response (Error 404): { "detail": "No shops available in the system" }

Get Shops by Owner

Path: /owner/{owner_id} Method: GET Description: Retrieves all shops owned by a specific owner

Path Parameters: owner_id: 5

Response (Success 200): [{ "shop_id": 101, "shop_name": "Elite Barber Studio", "address": "MG Road", "city": "Bangalore", "state": "Karnataka", "open_time": "09:00:00", "close_time": "21:00:00", "is_open": true }]

Response (Error 404): { "detail": "Owner not found or has no registered shops" }

Get Available Slots for Shop

Path: /shops/{shop_id}/slots/ Method: GET Description: Retrieves available barber slots for a specific date

Path Parameters: shop_id: 101

Query Parameters: date: 2025-10-22

Response (Success 200): [{ "slot_id": 501, "barber_id": 12, "barber_name": "Ravi Kumar", "slot_time": "10:00:00", "status": "available" }, { "slot_id": 502, "barber_id": 12, "barber_name": "Ravi Kumar", "slot_time": "11:00:00", "status": "available" }]

Response (Error 400): { "detail": "Invalid date format. Use YYYY-MM-DD" }

Response (Error 404): { "detail": "No slots found for the given shop or date" }

6.3 BARBER MANAGEMENT ENDPOINTS

Add Barber to Shop

Path: /barbers/add/{shop_id} Method: POST Description: Adds a new barber to the specified shop

Path Parameters: shop_id: 101

Request Body: { "barber_name": "Ravi Kumar", "start_time": "09:00:00", "end_time": "17:00:00", "is_available": true, "everyday": true }

Response (Success 200): { "message": "Barber added successfully", "barber_id": 12 }

Response (Error 404): { "detail": "Shop not found" }

Response (Error 400): { "detail": "Invalid time format" }

Update Barber Details

Path: /barbers/update/{barber_id} Method: PUT Description: Updates existing barber information

Path Parameters: barber_id: 12

Query Parameters: owner_id: 5

Request Body: { "barber_name": "Ravi Kumar Singh", "start_time": "10:00:00", "end_time": "18:00:00", "is_available": true, "everyday": true }

Response (Success 200): { "message": "Barber updated successfully", "barber_id": 12, "barber_name": "Ravi Kumar Singh" }

Response (Error 404): { "detail": "Barber not found" }

Response (Error 403): { "detail": "Unauthorized update attempt" }

Delete Barber

Path: /barbers/delete/{barber_id} Method: DELETE Description: Deletes barber and all associated records

Path Parameters: barber_id: 12

Query Parameters: owner_id: 5

Response (Success 200): { "message": "Barber and related data have been deleted successfully" }

Response (Error 404): { "detail": "Barber not found" }

Response (Error 403): { "detail": "Unauthorized deletion attempt" }

Get Available Barbers

Path: /barbers/available/{shop_id} Method: GET Description: Retrieves all available barbers for a shop

Path Parameters: shop_id: 101

Response (Success 200): [{ "barber_id": 12, "barber_name": "Ravi Kumar", "start_time": "09:00:00", "end_time": "17:00:00", "is_available": true }, { "barber_id": 13, "barber_name": "Arjun Singh", "start_time": "10:00:00", "end_time": "18:00:00", "is_available": true }]

Response (Error 404): { "detail": "No available barbers found for the given shop" }

6.4 SLOT GENERATION ENDPOINTS

Generate Barber Slots

Path: /generate-slots Method: POST Description: Triggers automatic slot generation for eligible barbers

Query Parameters (Optional): barber_id: 12 (Leave empty to generate for all eligible barbers)

Request Body: { "single_barber_id": 12 }

Response (Success 200): { "message": "Slots generated successfully", "barbers_processed": 1, "slots_created": 8 }

Response (Error 500): { "detail": "Error during slot generation" }

6.5 BOOKING ENDPOINTS

Book Barber Slots

Path: /book-slots/ Method: POST Description: Books one or multiple slots with a specific barber

Request Body: { "user_id": 7, "barber_id": 12, "shop_id": 101, "slot_ids": [501, 502, 503] }

Response (Success 200): { "message": "3 slots booked successfully", "user_id": 7, "barber_id": 12, "shop_id": 101, "booked_slots": [{ "slot_id": 501, "slot_date": "2025-10-22", "slot_time": "10:00:00", "status": "booked" }, { "slot_id": 502, "slot_date": "2025-10-22", "slot_time": "11:00:00", "status": "booked" }, { "slot_id": 503, "slot_date": "2025-10-22", "slot_time": "12:00:00", "status": "booked" }] }

Response (Error 400): { "detail": "Slot already booked" }

Response (Error 404): { "detail": "Slot not found" }

Response (Error 400): { "detail": "Invalid user, barber, or shop ID" }

---

7. MODULE INTERACTIONS

7.1 User Registration to Booking Flow

Step 1: User Registration Client calls POST /register with user credentials. User service validates input, checks for duplicates, hashes password, and stores user record in database.

Step 2: Email Verification Client calls POST /send-verification-otp with email. User service generates six-digit OTP and sends via email service. Client receives OTP and calls POST /verify-email. User service validates OTP, checks expiration, marks email as verified.

Step 3: User Login Client calls POST /login with email and password OR calls POST /otp-login for passwordless entry. User service validates credentials and returns user ID and role.

Step 4: Shop Discovery User calls GET /shops/ to view all available barbershops. Shop service retrieves all active shops from database and returns with details and operating hours.

Step 5: Slot Viewing User calls GET /shops/{shop_id}/slots/ with desired date. Shop service queries available slots for that shop and date, returns list of barbers and available times.

Step 6: Slot Booking User calls POST /book-slots/ with selected slot IDs. Booking service validates each slot, checks availability, marks as booked, creates booking records, and returns confirmation with details.

7.2 Owner Shop Setup Flow

Step 1: Owner Registration and Verification Owner completes user registration with role set to owner and verifies email address.

Step 2: Shop Creation Owner calls POST /create (with owner_id query parameter) with shop details. Shop service validates owner exists, checks no duplicate shop, stores shop record in database.

Step 3: Barber Addition Owner calls POST /barbers/add/{shop_id} to add barbers. Barber service validates shop exists, stores barber record with working hours and generates daily flag set to true.

Step 4: Automatic Slot Generation System internally triggers slot generation process. Slot generation service queries all barbers with generate_daily flag enabled, validates shop is open, generates hourly slots based on working hours.

Step 5: Booking Availability Customers can now view shop, see available slots, and book appointments through standard booking flow.

7.3 Barber Slot Generation Workflow

Trigger Point Slot generation is triggered daily by system scheduler or can be manually triggered via API endpoint.

Process

1. Slot generation service initializes database session
2. Queries all barbers where generate_daily is true and is_available is true
3. For each eligible barber, validates shop status and operating hours
4. Generates one-hour slots from barber start_time to end_time
5. For each potential slot, checks if already exists to prevent duplicates
6. Creates new slots with status available and is_booked false
7. Commits all transactions to database
8. Logs all operations for audit trail
9. Catches and logs any errors without disrupting system

---

8. DATA MODELS AND SCHEMAS

8.1 User Schema (user_schema.py)

UserRegister Schema

- username: String, required
- email: String, required, must be valid email
- password: String, required, minimum 8 characters
- phone_number: String, optional
- role: String, required, values customer or owner

UserLogin Schema

- email: String, required
- password: String, required
- role: String, required

UserResponse Schema

- id: Integer
- username: String
- email: String
- phone_number: String
- role: String

OTPSchema

- email: String, required
- otp: String, required, exactly 6 digits
- role: String, required

8.2 Shop Schema (shop_schemas.py)

ShopCreate Schema

- shop_name: String, required
- address: String, required
- city: String, required
- state: String, required
- open_time: Time, required, format HH:MM:SS
- close_time: Time, required, format HH:MM:SS

ShopResponse Schema

- shop_id: Integer
- shop_name: String
- address: String
- city: String
- state: String
- open_time: Time
- close_time: Time
- is_open: Boolean

SlotResponse Schema

- slot_id: Integer
- barber_id: Integer
- barber_name: String
- slot_time: Time
- status: String, values available or booked

8.3 Barber Schema (barber_schemas.py)

BarberCreate Schema

- barber_name: String, required
- start_time: Time, required, format HH:MM:SS
- end_time: Time, required, format HH:MM:SS
- is_available: Boolean, default true
- everyday: Boolean, default false

BarberUpdate Schema

- barber_name: String, optional
- start_time: Time, optional
- end_time: Time, optional
- is_available: Boolean, optional
- everyday: Boolean, optional

BarberResponse Schema

- barber_id: Integer
- barber_name: String
- start_time: Time
- end_time: Time
- is_available: Boolean
- shop_id: Integer

8.4 Booking Schema

BookingRequest Schema

- user_id: Integer, required
- barber_id: Integer, required
- shop_id: Integer, required
- slot_ids: List of integers, required, minimum 1 slot

BokedSlot Schema

- slot_id: Integer
- slot_date: Date, format YYYY-MM-DD
- slot_time: Time, format HH:MM:SS
- status: String, value booked

BookingResponse Schema

- message: String
- user_id: Integer
- barber_id: Integer
- shop_id: Integer
- booked_slots: List of BookedSlot objects

---

9. SERVICE LAYER ARCHITECTURE

9.1 Service Components

All business logic is contained within service modules that interact directly with SQLAlchemy database sessions:

User Service (user_service.py) Methods:

- register_user(db, user_data): Validates and registers new user
- send_verification_otp(db, email, role): Generates and sends OTP
- verify_email_otp(db, email, otp, role): Validates OTP and marks email verified
- login_user(db, email, password, role): Authenticates and validates credentials
- send_otp_login(db, email, role): Sends OTP for passwordless login
- verify_otp_login(db, email, otp, role): Validates login OTP
- get_user_details(db, email): Retrieves user profile information

Shop Service (shop_service.py) Methods:

- create_shop(db, owner_id, shop_data): Creates new shop with validation
- get_all_shops(db): Retrieves all active shops
- get_shops_by_owner(db, owner_id): Returns shops owned by specific owner
- get_available_slots(db, shop_id, date): Fetches available slots for date

Barber Service (barber_service.py) Methods:

- add_barber(db, shop_id, barber_data): Adds new barber to shop
- update_barber(db, barber_id, owner_id, barber_data): Updates barber details with authorization
- delete_barber(db, barber_id, owner_id): Deletes barber and cascading records
- get_available_barbers(db, shop_id): Returns available barbers in shop

Booking Service (booking_service.py) Methods:

- book_slots(db, user_id, barber_id, shop_id, slot_ids): Books multiple slots with validation

Slot Generation Service Methods:

- generate_barber_slots(db, single_barber_id): Generates hourly slots for eligible barbers

---

9.2 Service Transaction Management

Each service method:

- Accepts SQLAlchemy session as parameter
- Performs business logic validation
- Executes database operations
- Commits transaction on success
- Rolls back on error
- Returns formatted response
- Raises appropriate exceptions for error handling

---

10. UTILITIES AND HELPERS

10.1 Utility Components

Email Utilities (utils)

- send_otp_email(email, otp): Sends OTP to user email using configured SMTP
- send_verification_email(email): Sends verification link to email

Logging Utilities (utils)

- Centralized logging configuration
- Log rotation setup
- Log level management
- Audit trail recording for all significant operations

Helper Functions (utils)

- Date and time formatting
- Password hashing and verification
- OTP generation
- Data validation helpers

---

11. SECURITY FEATURES

11.1 Authentication Security

- Passwords are hashed using industry-standard algorithms before storage
- OTP codes are temporary and expire after five minutes
- Role-based access control on all operations
- Owner authorization checks before allowing shop or barber modifications

11.2 Data Validation

- Pydantic schemas enforce type checking and format validation
- Email format validation on all email inputs
- Time format validation for all schedule inputs
- Date format validation for slot queries

11.3 Audit Logging

- All user registrations logged with timestamps
- Login attempts recorded for security monitoring
- Booking creation logged with user and slot details
- Shop and barber modifications tracked with owner information
- Errors and exceptions logged for investigation

---

12. CONCLUSION

The Barber Booking System follows industry best practices with a clean, two-layer architecture that separates API routing from business logic. Direct database access through SQLAlchemy ORM eliminates the need for a separate repository layer while maintaining code organization and maintainability.

Each module has a clear and distinct responsibility. Routes handle HTTP communication, services handle business logic, and schemas handle data validation. This clear separation of concerns ensures that the system can be easily understood, maintained, and extended.

The system is designed for scalability with features including automatic slot generation, transaction management for data consistency, comprehensive error handling, and detailed audit logging. New modules can be added by following the established pattern of routes, services, and schemas without requiring restructuring of existing code.

Security is implemented throughout with password hashing, OTP verification, role-based access control, and detailed audit trails for compliance and monitoring purposes.

---

14. API TESTING AND USAGE EXAMPLES

14.1 Complete User Registration Flow Example

Step 1: Register User

Endpoint: POST /register

Body:

```
{
  "username": "Rajesh Kumar",
  "email": "rajesh@example.com",
  "password": "SecurePass123",
  "phone_number": "9876543210",
  "role": "customer"
}
```

Expected Response: 200

```
{
  "msg": "User registered successfully"
}
```

Step 2: Send Verification OTP

Endpoint: POST /send-verification-otp

Body:

```
{
  "email": "rajesh@example.com",
  "role": "customer"
}
```

Expected Response: 200

```
{
  "msg": "Verification OTP sent"
}
```

User receives OTP code via email

Step 3: Verify Email

Endpoint: POST /verify-email

Body:

```
{
```

```
 "email": "rajesh@example.com",
 "otp": "123456",
 "role": "customer"
}
```


Expected Response: 200


```
{
 "msg": "Email verified successfully"
}
```


Step 4: Login

Endpoint: POST /login

Body:


```
{
 "email": "rajesh@example.com",
 "password": "SecurePass123",
 "role": "customer"
}
```


Expected Response: 200


```
{
 "msg": "Login successful",
 "user_id": 1,
 "role": "customer"
}
```

14.2 Complete Shop Owner Setup Flow Example

Step 1: Owner Registration

Endpoint: POST /register

Body:


```
{
 "username": "Sharma Barbershop",
 "email": "sharma@barbershop.com",
 "password": "OwnerPass456",
 "phone_number": "9876543211",
 "role": "owner"
}
```

Expected Response: 200

```
{
  "msg": "User registered successfully"
}
```

Step 2: Owner Email Verification

Endpoint: POST /send-verification-otp

Body:

```
{
  "email": "sharma@barbershop.com",
  "role": "owner"
}
```

Follow verification steps same as customer

Step 3: Create Shop

Endpoint: POST /create?owner_id=2

Body:

```
{
  "shop_name": "Sharma Elite Barbershop",
  "address": "Main Street 123",
  "city": "Bangalore",
  "state": "Karnataka",
  "open_time": "09:00:00",
  "close_time": "21:00:00"
}
```

Expected Response: 200

```
{
  "message": "Shop created successfully",
  "shop_id": 101
}
```

Step 4: Add First Barber

Endpoint: POST /barbers/add/101

Body:

```
{
  "barber_name": "Ravi Kumar",
```

"start_time": "09:00:00",
"end_time": "17:00:00",
"is_available": true,
"everyday": true
}
Expected Response: 200
{
"message": "Barber added successfully",
"barber_id": 12
}

Step 5: Add Second Barber

Endpoint: POST /barbers/add/101
Body:
{
"barber_name": "Arjun Singh",
"start_time": "10:00:00",
"end_time": "18:00:00",
"is_available": true,
"everyday": true
}
Expected Response: 200
{
"message": "Barber added successfully",
"barber_id": 13
}

Step 6: System Generates Slots Automatically

Endpoint: POST /generate-slots
Body:
{
"single_barber_id": null
}
Expected Response: 200
{
"message": "Slots generated successfully",

```
  "barbers_processed": 2,
  "slots_created": 16
}
```


Note: System generates 8 slots per barber based on working hours

---

14.3 Complete Customer Booking Flow Example

Step 1: Customer Browsing Shops

Endpoint: GET /shops/

Expected Response: 200


```
[
  {
    "shop_id": 101,
    "shop_name": "Sharma Elite Barbershop",
    "address": "Main Street 123",
    "city": "Bangalore",
    "state": "Karnataka",
    "open_time": "09:00:00",
    "close_time": "21:00:00",
    "is_open": true
  }
]
```


Step 2: View Available Slots

Endpoint: GET /shops/101/slots/?date=2025-10-25

Expected Response: 200


```
[
  {
    "slot_id": 501,
    "barber_id": 12,
    "barber_name": "Ravi Kumar",
    "slot_time": "09:00:00",
    "status": "available"
  },
  {
    "slot_id": 502,
```

```
  "barber_id": 12,
  "barber_name": "Ravi Kumar",
  "slot_time": "10:00:00",
  "status": "available"
 },
 {
  "slot_id": 503,
  "barber_id": 13,
  "barber_name": "Arjun Singh",
  "slot_time": "10:00:00",
  "status": "available"
 }
]
```


Step 3: Customer Books Slots

Endpoint: POST /book-slots/

Body:


```
{
 "user_id": 1,
 "barber_id": 12,
 "shop_id": 101,
 "slot_ids": [501, 502]
}
```


Expected Response: 200


```
{
 "message": "2 slots booked successfully",
 "user_id": 1,
 "barber_id": 12,
 "shop_id": 101,
 "booked_slots": [
  {
   "slot_id": 501,
   "slot_date": "2025-10-25",
   "slot_time": "09:00:00",
   "status": "booked"
  },
```

```
  {
    "slot_id": 502,
    "slot_date": "2025-10-25",
    "slot_time": "10:00:00",
    "status": "booked"
  }
 ]
}
```

14.4 Barber Management Operations Example

Update Barber Details

Endpoint: PUT /barbers/update/12?owner_id=2

Body:


```
{
 "barber_name": "Ravi Kumar Singh",
 "start_time": "10:00:00",
 "end_time": "18:00:00",
 "is_available": true,
 "everyday": true
}
```


Expected Response: 200


```
{
 "message": "Barber updated successfully",
 "barber_id": 12,
 "barber_name": "Ravi Kumar Singh"
}
```


Get Available Barbers

Endpoint: GET /barbers/available/101

Expected Response: 200


```
[
 {
  "barber_id": 12,
  "barber_name": "Ravi Kumar Singh",
  "start_time": "10:00:00",
```

```
  "end_time": "18:00:00",
  "is_available": true
 },
 {
  "barber_id": 13,
  "barber_name": "Arjun Singh",
  "start_time": "10:00:00",
  "end_time": "18:00:00",
  "is_available": true
 }
]
```


Delete Barber

Endpoint: DELETE /barbers/delete/13?owner_id=2

Expected Response: 200


```
{
 "message": "Barber and related data have been deleted successfully"
}
```


---

15. ERROR HANDLING AND RESPONSES

15.1 HTTP Status Codes Used

200 OK Request successful, response contains requested data or confirmation.

201 Created Resource successfully created, contains details of new resource.

400 Bad Request Invalid input, missing required fields, or business logic violation. Response contains error details.

401 Unauthorized Authentication failed, invalid credentials, or invalid OTP.

403 Forbidden User lacks permission to perform action, typically authorization failure.

404 Not Found Requested resource does not exist in database.

500 Internal Server Error Unexpected server error during processing, typically database or system errors.

15.2 Error Response Format

All error responses follow consistent format:

```
{
  "detail": "Descriptive error message explaining what went wrong"
}
```

15.3 Common Error Scenarios

Email Already Registered

Status: 400

```
{
  "detail": "Email already registered"
}
```

Invalid OTP

Status: 400

```
{
  "detail": "Invalid or expired OTP"
}
```

Unauthorized Operation

Status: 403

```
{
  "detail": "Unauthorized access. Only shop owner can perform this action"
}
```

Slot Already Booked

Status: 400

```
{
  "detail": "Slot already booked"
}
```

Resource Not Found

Status: 404

```
{
```

```
  "detail": "User not found"
}
```

---

16. PERFORMANCE CONSIDERATIONS

16.1 Database Optimization

- Indexes on frequently queried fields like email, user_id, shop_id
- Efficient slot queries filtered by date to avoid full table scans
- Transaction batching for multiple slot bookings
- Connection pooling for database performance

16.2 Slot Generation Optimization

- Slots generated in batches for multiple barbers
- Duplicate prevention through existence checks before insertion
- Past slot filtering to avoid unnecessary data
- Configurable single barber or batch generation options

16.3 Query Optimization

- Specific field selection in responses to minimize data transfer
- Lazy loading of related objects only when needed
- Filtering at database level rather than in application code
- Proper pagination for large result sets

---

17. MAINTENANCE AND MONITORING

17.1 Logging Strategy

All significant operations logged including:

- User registration and authentication attempts
- Shop and barber modifications
- Booking operations and slot changes
- Slot generation process and results
- All errors and exceptions with stack traces

17.2 Log Location

- Application logs stored in app/logs directory

- Log rotation configured to manage disk space
- Different log levels for development and production environments

17.3 Monitoring Points

- Monitor OTP generation and verification success rates
- Track booking success and failure rates
- Monitor slot generation completion and duration
- Track database query performance
- Monitor API response times

18. FUTURE ENHANCEMENTS

The current architecture supports easy addition of new features:

Possible Enhancements

- Payment integration for booking confirmations
- Email reminders before appointments
- Customer ratings and reviews
- Barber schedule preferences
- Recurring appointment bookings
- SMS notifications in addition to email
- Mobile app authentication tokens
- API rate limiting and throttling
- Advanced analytics and reporting
- Cancellation and rescheduling functionality