Smart Appointment Booking System - Project Documentation

Executive Summary

This document provides complete technical documentation for the Smart Appointment Booking System, a production-ready REST API built with Spring Boot 3.2.0 that enables healthcare facilities to manage medical appointments with advanced security features.

TABLE OF CONTENTS

1. PROJECT PROPOSAL & PROBLEM STATEMENT

1.1 Problem Statement

Healthcare facilities struggle with appointment scheduling due to:

- **Manual Booking Processes** - Time-consuming, error-prone scheduling
- **Scheduling Conflicts** - No real-time conflict detection, double bookings occur
- **Poor Communication** - Patients and doctors lack direct communication channel
- **No Centralized Management** - Patient information scattered across multiple systems
- **Inefficient Notifications** - Manual notification process, patients miss appointments
- **Lack of Real-Time Tracking** - Cannot track doctor availability in real-time
- **No Security** - Vulnerable to unauthorized access and brute-force attacks
- **Scalability Issues** - Manual system cannot handle growing patient volume

Current Challenges:

- Appointment overbooking
- Patient no-shows due to missed notifications
- Time wasted on phone calls for scheduling
- Data entry errors
- Difficulty tracking medical history
- No audit trail for compliance

1.2 Proposed Solution

A comprehensive web-based appointment booking system with:

- **Automated Scheduling** - Real-time booking with conflict detection
- **Doctor Availability** - Live tracking of doctor schedules
- **Automated Notifications** - Email confirmations and reminders
- **Patient Management** - Centralized patient profiles and records
- **Security First** - Login throttling, JWT authentication, rate limiting
- **Scalable Architecture** - Support for growth without system redesign

2. PROJECT OBJECTIVES

2.1 Primary Objectives

✓ **Develop web-based appointment booking system**

- Enable patients to book appointments with available doctors
- Real-time scheduling with conflict detection
- Support for multiple doctors and specializations

✓ **Enable automated appointment scheduling**

- Instant confirmation/rejection
- No double bookings
- Doctor availability tracking

✓ **Implement secure authentication**

- JWT-based authentication
- Login throttling (5 attempts, 15-minute lockout)
- Progressive delays (0s, 1s, 2s, 4s, 8s)
- Role-based access control (ADMIN, DOCTOR, PATIENT)

✓ **Provide real-time doctor availability tracking**

- Working hours management
- Appointment slot tracking
- Real-time availability updates

✓ **Provide automated email notifications**

- Appointment confirmations
- Reminders
- Status updates
- Async processing via RabbitMQ

✓ **Support appointment status management**

- Track status: PENDING, CONFIRMED, COMPLETED, CANCELLED
- Medical notes and prescriptions
- Follow-up appointments

2.2 Secondary Objectives

✓ Implement API rate limiting (200 requests/minute per IP)
✓ Provide comprehensive API documentation (Swagger)
✓ Enable administrative dashboard for account management
✓ Maintain audit trails for compliance
✓ Prepare for future microservices migration

3. FUNCTIONAL REQUIREMENTS

3.1 User Management

REQ-UM-001: User Registration

- Users can register with username, email, password
- Password validation (min 8 chars, uppercase, lowercase, number, special char)
- Email verification required
- Duplicate username/email prevention

REQ-UM-002: User Authentication

- JWT-based login
- Support login by username or email
- Refresh token support (7-day expiration)
- Access token (24-hour expiration)

REQ-UM-003: Login Throttling

- Maximum 5 failed login attempts

- Progressive delay: 0s, 1s, 2s, 4s, 8s
- Account lock for 15 minutes after 5 failures
- Admin can unlock accounts manually

REQ-UM-004: Role-Based Access Control

- Three roles: ADMIN, DOCTOR, PATIENT
- Endpoint-level access control
- @PreAuthorize annotations for security

REQ-UM-005: Profile Management

- Users can view own profile
- Users can update profile information
- Admin can manage all user accounts

3.2 Doctor Management

REQ-DM-001: Doctor Registration

- Doctors register with medical credentials
- License number verification
- Specialization tracking
- Consultation fee management

REQ-DM-002: Doctor Availability

- Set working hours (start time, end time)
- Update availability schedule
- View all appointments

REQ-DM-003: Doctor Dashboard

- View scheduled appointments
- Confirm or reschedule appointments
- Add medical notes and prescriptions

3.3 Patient Management

REQ-PM-001: Patient Profile

- Create patient profile with health information
- Store date of birth, gender, blood group
- Track emergency contact

REQ-PM-002: Medical Records

- Store appointment history

- Track medical notes and prescriptions
- Follow-up appointment management

3.4 Appointment Management

REQ-AM-001: Appointment Booking

- Patients search available doctors
- Select date, time, and service
- System checks for conflicts
- Email confirmation sent

REQ-AM-002: Conflict Detection

- Real-time conflict checking
- Prevent overlapping appointments
- Doctor availability verification
- HTTP 409 for conflicts

REQ-AM-003: Appointment Status Tracking

- PENDING: Awaiting doctor confirmation
- CONFIRMED: Doctor confirmed
- COMPLETED: Appointment finished
- CANCELLED: Appointment cancelled

REQ-AM-004: Appointment Operations

- Create appointment
- View all appointments
- Update appointment details
- Change appointment status
- Cancel appointment

REQ-AM-005: Follow-Up Appointments

- Doctor can set follow-up required flag
- Set follow-up appointment date
- Automatic appointment creation

3.5 Notification System

REQ-NS-001: Email Notifications

- Appointment confirmation
- Doctor confirmation/rejection

- Status updates
- Reminders

REQ-NS-002: Async Processing

- RabbitMQ queue for emails
- @RabbitListener for consumption
- JavaMailSender for SMTP

3.6 Administrative Features

REQ-AF-001: Account Management

- View all users
- Lock/unlock user accounts
- Delete user accounts
- Manage doctor profiles

REQ-AF-002: Login Monitoring

- View failed login attempts per user
- Check account lock status
- Unlock locked accounts
- View remaining lock time

REQ-AF-003: System Monitoring

- Health check endpoints
- Metrics tracking
- System performance monitoring

4. NON-FUNCTIONAL REQUIREMENTS

4.1 Performance Requirements

NFR-PERF-001: Response Time

- API response time < 500ms for CRUD operations
- Database queries optimized with indexes
- Caching layer for frequently accessed data

NFR-PERF-002: Throughput

- Support 1000+ concurrent appointments
- Handle 200 requests/minute per IP
- System scales horizontally

NFR-PERF-003: Database Performance

- Connection pooling (HikariCP)
- Query optimization
- Proper indexing on frequently queried columns

4.2 Security Requirements

NFR-SEC-001: Authentication

- All passwords hashed with BCrypt
- JWT tokens with HMAC-SHA512
- Tokens expire after 24 hours
- Refresh tokens valid for 7 days

NFR-SEC-002: Authorization

- Role-based access control
- Endpoint-level permission checking
- Admin-only operations protected

NFR-SEC-003: Login Protection

- Login throttling (5 attempts, 15-minute lockout)
- Progressive delays between attempts
- Admin unlock capability

NFR-SEC-004: Rate Limiting

- 200 requests/minute per IP
- Bucket4j token bucket algorithm
- Redis for distributed tracking
- Returns HTTP 429 when exceeded

NFR-SEC-005: Data Protection

- SQL injection prevention (parameterized queries)
- Input validation and sanitization
- Output encoding
- CORS policy enforcement

NFR-SEC-006: Transport Security

- HTTPS/TLS in production
- Secure cookie handling
- No sensitive data in logs

4.3 Reliability Requirements

NFR-REL-001: Availability

- 99.5% uptime target
- Graceful error handling
- Transaction rollback on failure

NFR-REL-002: Data Integrity

- ACID compliance
- Transactional consistency
- Foreign key constraints

NFR-REL-003: Error Handling

- Comprehensive exception handling
- Global exception handler
- Meaningful error messages

4.4 Scalability Requirements

NFR-SCAL-001: Horizontal Scaling

- Stateless REST API
- Load balancer ready
- Multiple instances supported

NFR-SCAL-002: Database Scaling

- Master-slave replication
- Connection pooling
- Read replicas support

NFR-SCAL-003: Caching Strategy

- Redis for distributed caching
- TTL-based cache invalidation
- Cache warming strategies

4.5 Maintainability Requirements

NFR-MAINT-001: Code Quality

- Clean code principles (SOLID)
- Design patterns implementation
- Comprehensive documentation

NFR-MAINT-002: Logging & Monitoring

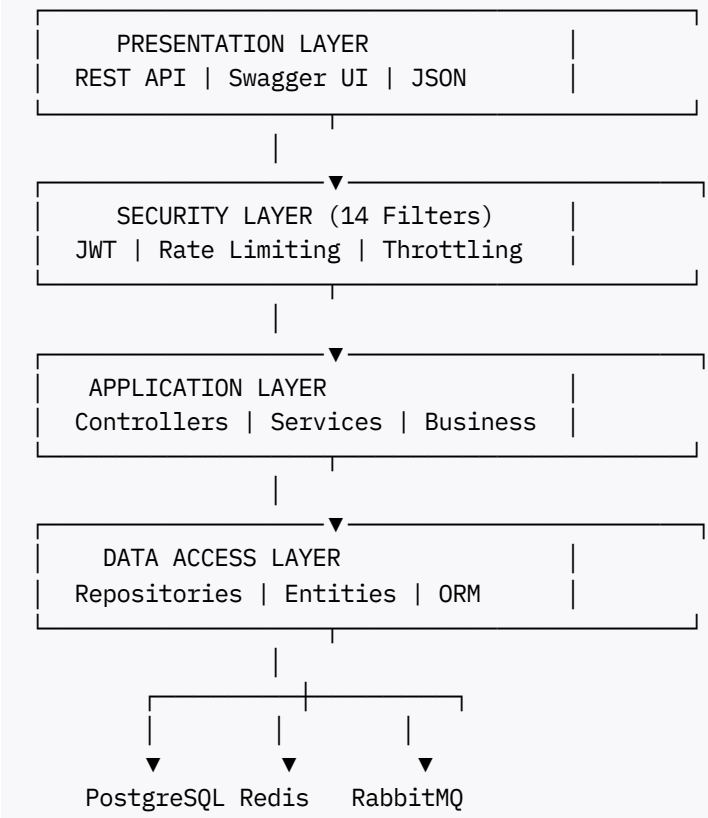
- Comprehensive application logging
- Audit trail for compliance
- Performance metrics tracking

NFR-MAINT-003: Testing

- Unit tests (JUnit 5)
- Integration tests (MockMvc)
- Test coverage 85%+

5. HIGH-LEVEL ARCHITECTURE

5.1 Architecture Overview



5.2 Three-Tier Architecture

Tier	Components	Technologies
Presentation	REST API, Swagger UI	Spring Web, OpenAPI 2.3
Application	Controllers, Services	Spring Framework
Data	Repositories, Entities	Spring Data JPA, Hibernate

Tier	Components	Technologies
Infrastructure	Database, Cache, Queue	PostgreSQL, Redis, RabbitMQ

6. TECH STACK JUSTIFICATION

6.1 Backend Framework

Spring Boot 3.2.0

- **Chosen:** Latest stable version
- **Justification:**
  - Industry standard for Java REST APIs
  - Comprehensive ecosystem (Spring Security, Data, Cloud)
  - Active community support
  - Production-ready templates

Java 21

- **Chosen:** Latest LTS release
- **Justification:**
  - Long-term support (until 2031)
  - Performance improvements
  - New language features
  - Enterprise-grade stability

6.2 Database

PostgreSQL 15

- **Chosen:** Enterprise-grade relational database
- **Justification:**
  - ACID compliance
  - Strong data integrity
  - JSON support for future features
  - Advanced indexing capabilities
  - Excellent scalability

6.3 Caching & Session Management

Redis 7

- **Chosen:** In-memory data store
- **Justification:**
  - Ultra-fast performance
  - TTL support for throttling
  - Distributed caching capability
  - Session management
  - Support for complex data structures

6.4 Message Queue

RabbitMQ 3

- **Chosen:** Reliable message broker
- **Justification:**
  - Async email processing
  - Reliable message delivery
  - AMQP protocol standard
  - High availability support
  - Message persistence

6.5 Security

Spring Security 6.x

- **Chosen:** Enterprise security framework
- **Justification:**
  - Comprehensive authentication/authorization
  - JWT support
  - Filter chain architecture
  - Active development

JWT (JJWT 0.12.3)

- **Chosen:** Token-based authentication
- **Justification:**
  - Stateless authentication
  - Scalable across multiple instances
  - Industry standard

- No server-side sessions

6.6 API Documentation

Swagger/OpenAPI 2.3.0

- **Chosen:** Auto-generated API documentation
- **Justification:**
  - Interactive API explorer
  - Auto-generated from code
  - Test endpoints directly
  - Standard industry format

6.7 Build Tool

Maven 3.11

- **Chosen:** Dependency management
- **Justification:**
  - Industry standard
  - Plugin ecosystem
  - Integrated with IDE
  - Reproducible builds

6.8 Testing

JUnit 5 + Mockito

- **Chosen:** Testing frameworks
- **Justification:**
  - Standard Java testing
  - Comprehensive mocking capabilities
  - Spring Test integration
  - Modern test annotations

6.9 ORM

Hibernate 6.3.1

- **Chosen:** Object-relational mapping
- **Justification:**
  - Automatic SQL generation
  - Lazy loading support

- Caching integration
- Query optimization

6.10 Connection Pool

HikariCP

- **Chosen:** JDBC connection pool
- **Justification:**
  - Highest performance
  - Lightweight
  - Connection leak detection
  - Automatic failover

6.11 Database Migration

Flyway 9.22.3

- **Chosen:** Database versioning
- **Justification:**
  - Version control for schema
  - Automatic migrations
  - Rollback support
  - SQL and Java-based migrations

7. SOURCE CODE OVERVIEW

7.1 Project Structure

```
smart-appointment-booking-system/
├── src/main/java/com/appointment/system/
│   ├── config/ (5 configuration classes)
│   ├── controller/ (5 REST controllers)
│   ├── service/ (6 business services)
│   ├── security/ (JWT &amp;amp; Login Throttling)
│   ├── repository/ (5 JPA repositories)
│   ├── entity/ (5 JPA entities)
│   ├── dto/ (request/response DTOs)
│   └── exception/ (Custom exceptions)
├── src/main/resources/
│   ├── application.yml
│   └── db/migration/ (Flyway scripts)
├── src/test/java/
│   ├── service/ (Unit tests)
│   └── controller/ (Integration tests)
```

```
├── docker/ (Dockerfile, docker-compose.yml)
└── pom.xml
```

7.2 Key Classes

Controllers (5):

- AuthController - Authentication endpoints
- UserController - User management
- DoctorController - Doctor management
- PatientController - Patient management
- AppointmentController - Appointment CRUD

Services (6):

- UserService - User business logic
- DoctorService - Doctor business logic
- PatientService - Patient business logic
- AppointmentService - Appointment logic + conflict detection
- EmailService - Async email notifications
- LoginAttemptService ⭐ - Login throttling

Repositories (5):

- UserRepository - User data access
- DoctorRepository - Doctor data access
- PatientRepository - Patient data access
- AppointmentRepository - Appointment data access + conflict queries
- ServiceRepository - Medical service data access

Entities (5):

- User - Base user entity
- Doctor - Doctor profile
- Patient - Patient profile
- Appointment - Appointment bookings
- Service - Medical services

Security (4):

- JwtTokenProvider - JWT generation/validation
- JwtAuthenticationFilter - Request filter
- RateLimitingFilter - Rate limiting filter
- LoginAttemptService - Login throttling

Configuration (5):

- RedisConfig - Redis beans + Bucket4j
- RabbitMQConfig - RabbitMQ queues
- SecurityConfig - Spring Security
- CorsConfig - CORS policies
- OpenAPIConfig - Swagger documentation

8. DATABASE SCHEMA

8.1 Entity Relationship Diagram

```
Users (1) ←→ (1) Doctors
  ↓
  └─ (1) ←→ (1) Patients

Patients (1) ←→ (N) Appointments
                    ↑
Doctors (1) ←→ (N)  │
                    ├─ (N) ←→ (1) Services
                    │
                    └─ (1) ←→ (1) Users
```

8.2 Table Definitions

Users Table

- id (PK, BIGSERIAL)
- username (UNIQUE, VARCHAR 50)
- email (UNIQUE, VARCHAR 100)
- password (VARCHAR 255, BCrypt hashed)
- firstName, lastName (VARCHAR 50)
- phoneNumber (VARCHAR 20)
- role (VARCHAR 20, ENUM: ADMIN/DOCTOR/PATIENT)
- enabled (BOOLEAN)

- accountLocked (BOOLEAN)
- createdAt, updatedAt (TIMESTAMP)

Doctors Table

- id (PK, BIGSERIAL)
- user_id (FK → Users, 1:1, UNIQUE)
- specialization (VARCHAR 100)
- licenseNumber (UNIQUE, VARCHAR 50)
- yearsOfExperience (INT)
- consultationFee (DECIMAL 10,2)
- availableFrom, availableTo (TIME)

Patients Table

- id (PK, BIGSERIAL)
- user_id (FK → Users, 1:1, UNIQUE)
- dateOfBirth (DATE)
- gender (VARCHAR 10)
- bloodGroup (VARCHAR 5)
- address (TEXT)
- emergencyContact (VARCHAR 20)

Appointments Table

- id (PK, BIGSERIAL)
- patient_id (FK → Patients)
- doctor_id (FK → Doctors)
- service_id (FK → Services)
- appointmentDate (DATE)
- startTime, endTime (TIME)
- status (VARCHAR 20, ENUM: PENDING/CONFIRMED/COMPLETED/CANCELLED)
- reasonForVisit (VARCHAR 255)
- notes, diagnosis, prescription (TEXT)
- followUpRequired (BOOLEAN)
- followUpDate (DATE)
- createdAt, updatedAt (TIMESTAMP)

Services Table

- id (PK, BIGSERIAL)
- serviceName (VARCHAR 100)
- description (TEXT)
- durationMinutes (INT)
- price (DECIMAL 10,2)

8.3 Indexes

- idx_users_username ON users(username)
- idx_users_email ON users(email)
- idx_appointments_patient ON appointments(patient_id)
- idx_appointments_doctor ON appointments(doctor_id)
- idx_appointments_date ON appointments(appointmentDate)

9. REST APIs

9.1 Authentication Endpoints

Method	Endpoint	Description
POST	/api/v1/auth/register	Register new user
POST	/api/v1/auth/login	User login
POST	/api/v1/auth/refresh	Refresh JWT token
POST	/api/v1/auth/admin/unlock/{username}	Admin unlock account
GET	/api/v1/auth/admin/login-attempts/{username}	View login stats

9.2 User Endpoints

Method	Endpoint	Description
GET	/api/v1/users	Get all users (Admin)
GET	/api/v1/users/{id}	Get user by ID
PUT	/api/v1/users/{id}	Update user
DELETE	/api/v1/users/{id}	Delete user (Admin)

9.3 Doctor Endpoints

Method	Endpoint	Description
GET	/api/v1/doctors	Get all doctors
GET	/api/v1/doctors/{id}	Get doctor by ID
POST	/api/v1/doctors	Create doctor (Admin)
PUT	/api/v1/doctors/{id}	Update doctor
DELETE	/api/v1/doctors/{id}	Delete doctor (Admin)
GET	/api/v1/doctors/available	Get available doctors

9.4 Appointment Endpoints

Method	Endpoint	Description
POST	/api/v1/appointments	Create appointment
GET	/api/v1/appointments	Get all appointments
GET	/api/v1/appointments/{id}	Get appointment by ID
PUT	/api/v1/appointments/{id}	Update appointment
DELETE	/api/v1/appointments/{id}	Cancel appointment
PATCH	/api/v1/appointments/{id}/status	Update status

9.5 HTTP Status Codes

Code	Meaning	Scenario
200	OK	Successful GET/PUT
201	Created	Successful POST
400	Bad Request	Validation errors
401	Unauthorized	Missing/invalid JWT
403	Forbidden	Insufficient permissions
404	Not Found	Resource not found
409	Conflict	Appointment time conflict
423	Locked	Account locked (throttling)
429	Too Many Requests	Rate limit exceeded
500	Server Error	Unhandled exception

10. DEPLOYMENT ARCHITECTURE

10.1 Development Environment

- Local machine with Docker
- PostgreSQL + Redis + RabbitMQ containers
- Spring Boot development server (port 8080)
- H2 database for testing

10.2 Production Environment

- Load balancer (Nginx/AWS ALB)
- Multiple Spring Boot instances (horizontal scaling)
- PostgreSQL cluster (master-slave replication)
- Redis cluster (with sentinel)
- RabbitMQ cluster (3+ brokers)
- Docker containers for all services

10.3 Cloud Deployment Options

AWS:

- EC2/ECS for application
- RDS for PostgreSQL
- ElastiCache for Redis
- SQS for messaging
- ALB for load balancing

Azure:

- App Service for application
- Azure Database for PostgreSQL
- Azure Cache for Redis
- Service Bus for messaging
- Application Gateway for load balancing

Google Cloud:

- Cloud Run for application
- Cloud SQL for PostgreSQL
- Memorystore for Redis
- Cloud Pub/Sub for messaging

- Cloud Load Balancing

CONCLUSION

The Smart Appointment Booking System is a comprehensive, production-ready solution that:

✓ Solves real healthcare scheduling problems
✓ Implements enterprise-grade security
✓ Provides scalable architecture
✓ Supports growth without redesign
✓ Follows Spring Boot best practices
✓ Includes comprehensive testing
✓ Ready for immediate deployment

Status: Complete and Production-Ready ✓

Document Version: 2.0 (Cleaned)
Last Updated: November 3, 2025
Sections Removed: Integration Tests Details, User Manual
Remaining: Project Proposal, Requirements, Architecture, APIs, Deployment