CLAUDE.md

This file provides guidance to Claude Code (claude.ai/code) when working with code in this repository.

Project Overview

This is a **Class Booking System** REST API built with Spring Boot 3.2.2, providing JWT-authenticated class scheduling and booking functionality with role-based access control (USER, ADMIN, INSTRUCTOR).

Key Technology: Java 17, Spring Boot, Spring Security with JWT, Spring Data JPA, H2 Database, Flyway migrations, Lombok

Codebase Knowledge Base

For detailed architecture and implementation knowledge without re-scanning the entire codebase, refer to:

• `ARCHITECTURE_KNOWLEDGE.md` - Comprehensive knowledge base with detailed class descriptions, patterns, and implementation details
• `ARCHITECTURE.md` - Complete architecture analysis and improvement recommendations
When starting a new Claude Code session, reading `ARCHITECTURE_KNOWLEDGE.md` first provides full context of the codebase structure, key classes, database schema, API endpoints, and architectural patterns.

Build and Run Commands

```
# Build the project mvn clean install # Run the application mvn
spring-boot:run # Run tests mvn test # Run tests with coverage mvn clean test
jacoco:report # Skip tests during build mvn clean install -DskipTests
```

Application runs on: `http://localhost:8080`

Development Resources

• **Swagger UI**: http://localhost:8080/swagger-ui.html
• **H2 Console**: http://localhost:8080/h2-console
• JDBC URL: `jdbc:h2:mem:bookingdb`
• Username: `sa`
• Password: (empty)

Architecture Overview

Three-Layer Architecture

```
Controller Layer (REST endpoints) ↓ Service Layer (business logic +
transactions) ↓ Repository Layer (JPA data access) ↓ H2 In-Memory Database
```

Package Structure

• `config/` - Security and OpenAPI configuration
• `controller/` - REST API endpoints (AuthController, UserController, ClassController, BookingController)
• `dto/request/` and `dto/response/` - Data transfer objects (Entity ↔ JSON isolation)

• `entity/` - JPA entities (User, Instructor, ClassSchedule, Booking)
• `repository/` - Spring Data JPA repositories
• `service/` - Business logic and transactional operations
• `security/` - JWT token provider, authentication filter, UserDetailsService
• `exception/` - Custom exceptions and global exception handler

Database Schema

Four main tables managed by Flyway migrations in `src/main/resources/db/migration/`:

1. **users** - User accounts with BCrypt password hashing
2. **instructors** - Instructor profiles (1-to-1 with users)
3. **class_schedules** - Class schedules with capacity management
4. **bookings** - Booking records (many-to-many between users and classes)

Critical Implementation Patterns

Concurrency Control for Bookings

The booking system uses pessimistic locking to prevent race conditions and overbooking.

```
// In ClassScheduleRepository @Lock(LockModeType.PESSIMISTIC_WRITE)
@Query("SELECT cs FROM ClassSchedule cs WHERE cs.id = :id")
Optional<ClassSchedule> findByIdWithLock(@Param("id") Long id);
```

When modifying booking logic:

• ALWAYS use `findByIdWithLock()` when reading class schedules for booking/cancellation
• Keep the pessimistic lock within the `@Transactional` boundary
• Update `currentBookings` counter atomically with the booking operation

JWT Authentication Flow

1. User registers/logs in via `/api/v1/auth/*` endpoints (no authentication required)
2. `AuthService` validates credentials and generates JWT token (24-hour expiration, HS256 algorithm)
3. JWT token is passed in subsequent requests: `Authorization: Bearer <token>`
4. `JwtAuthenticationFilter` intercepts requests, validates token, sets SecurityContext
5. Controllers access authenticated user via `@AuthenticationPrincipal UserDetails`

Security Configuration (`SecurityConfig.java`):

• CSRF disabled (stateless API)
• Session management: STATELESS
• Public endpoints: `/api/v1/auth/**`, `/h2-console/**`, `/swagger-ui/**`, `GET /api/v1/classes/**`
• Role-based access: ADMIN can manage all, INSTRUCTOR can manage classes, USER can book

Role Promotion (Development)

New users register as `ROLE_USER` by default. To test ADMIN/INSTRUCTOR features:

```
-- Access H2 Console and run: UPDATE users SET role = 'ROLE_ADMIN' WHERE email
= 'user@example.com'; UPDATE users SET role = 'ROLE_INSTRUCTOR' WHERE email =
'user@example.com';
```

DTO Pattern

Entities NEVER appear in Controller responses/requests. Always use:

• Request DTOs for input validation (`@Valid`, `@NotBlank`, `@Email`)

• Response DTOs for output serialization
• Conversion happens in Service layer (see `convertToResponse()` methods)
This isolates database structure from API contract.

Flyway Migrations

Database schema is versioned in `src/main/resources/db/migration/`:

• V1__create_users_table.sql
• V2__create_instructors_table.sql
• V3__create_class_schedules_table.sql
• V4__create_bookings_table.sql
• V5__add_version_column.sql
Never modify existing migrations. Create new migrations for schema changes (e.g., `V6__add_new_column.sql`).

`spring.jpa.hibernate.ddl-auto` is set to `validate` - Flyway manages all schema changes.

Common Development Workflows

Adding a New Endpoint

1. Create Request/Response DTOs in `dto/request/` and `dto/response/`
2. Add business logic method in appropriate Service class
3. Create Controller method with proper security annotations (`@PreAuthorize` if needed)
4. Update Swagger documentation (SpringDoc auto-generates from annotations)

Modifying Security Rules

Edit `SecurityConfig.java` `filterChain()` method:

• Use `requestMatchers()` for path-based rules
• Use `hasRole()` or `hasAnyRole()` for role-based restrictions
• Remember: Spring Security automatically prepends `ROLE_` prefix (use `ADMIN` not `ROLE_ADMIN` in code)

Testing API Endpoints

Use Swagger UI at http://localhost:8080/swagger-ui.html for interactive testing:

1. Register a user via `/api/v1/auth/register`
2. Copy the JWT token from response
3. Click "Authorize" button in Swagger UI
4. Enter: `Bearer <your-token>`
5. Test protected endpoints

Database Debugging

Access H2 Console at http://localhost:8080/h2-console:

• View current data state
• Test queries
• Manually update user roles for testing
• Data resets on every application restart (in-memory database)

Known Architectural Constraints

1. **H2 In-Memory Database**: Data is lost on restart. Not suitable for production. To migrate to PostgreSQL/MySQL, update `application.yml` datasource configuration and add appropriate

driver dependency to `pom.xml`.
2. **No Caching Layer**: All queries hit the database directly. For production, consider adding Spring Cache with Redis for frequently accessed data (class listings, user profiles).
3. **Synchronous Processing**: Email notifications and other async tasks would block request handling. Consider adding `@Async` or message queue (RabbitMQ/Kafka) for production.
4. **Field Injection**: Current code uses `@Autowired` on fields. Constructor injection is preferred for testability and immutability.
5. **Transaction Boundaries**: Some Service methods have broad `@Transactional` scope including DTO conversion. Consider splitting transaction logic from transformation logic for better performance.
See `ARCHITECTURE.md` for detailed analysis and improvement recommendations.

API Endpoint Summary

Public Endpoints

• `POST /api/v1/auth/register` - Register new user
• `POST /api/v1/auth/login` - Login (returns JWT)
• `GET /api/v1/classes` - List all classes (supports `?availableOnly=true`, `?status=SCHEDULED`)
• `GET /api/v1/classes/{id}` - Get class details

User Endpoints (Requires Authentication)

• `GET /api/v1/users/me` - Get current user profile
• `POST /api/v1/bookings` - Book a class
• `DELETE /api/v1/bookings/{id}` - Cancel booking
• `GET /api/v1/bookings/my-bookings` - Get user's bookings

Admin/Instructor Endpoints

• `POST /api/v1/classes` - Create class (ADMIN/INSTRUCTOR)
• `PUT /api/v1/classes/{id}` - Update class (ADMIN/INSTRUCTOR)
• `DELETE /api/v1/classes/{id}` - Cancel class (ADMIN/INSTRUCTOR)
• `GET /api/v1/users` - List all users (ADMIN only)
• `GET /api/v1/bookings` - List all bookings (ADMIN only)

Configuration Properties

Key settings in `application.yml`:

• `server.port`: 8080
• `jwt.secret`: JWT signing key (change in production)
• `jwt.expiration`: 86400000ms (24 hours)
• `spring.jpa.show-sql`: true (set false in production)
• `logging.level.com.booking.system`: DEBUG (set to INFO/WARN in production)

Quick Reference: Booking Flow

1. User registers → receives JWT token
2. User lists available classes → `GET /api/v1/classes?availableOnly=true`
3. User books class → `POST /api/v1/bookings` with `classScheduleId`
4. System checks:
• Class status is SCHEDULED
• Class hasn't started yet

• Capacity not exceeded (`currentBookings < capacity`)
• User hasn't already booked this class
5. If valid: Creates booking, increments `currentBookings`, returns confirmation
6. User can cancel → `DELETE /api/v1/bookings/{id}` (decrements `currentBookings`)
All booking operations are protected by database-level pessimistic locks to prevent double-booking in concurrent scenarios.

For Claude Code: Working with this Codebase

To efficiently work with this codebase in future sessions without re-scanning all files:

1. **First Read**: Start by reading `ARCHITECTURE_KNOWLEDGE.md` to understand the complete codebase structure, key classes, and implementation patterns.
2. **Architecture Reference**: Consult `ARCHITECTURE.md` for detailed analysis, improvement recommendations, and scalability planning.
3. **Key Patterns to Remember**:
• Concurrency: Pessimistic locking in `ClassScheduleRepository.findByIdWithLock()`
• Security: JWT authentication flow with 24-hour expiration
• Data Flow: Controller → Service → Repository → Database
• DTO Pattern: Entities never exposed directly in API responses
4. **Common Development Tasks**:
• Adding endpoints: Create DTOs → Add Service logic → Add Controller method
• Security changes: Modify `SecurityConfig.java` filter chain
• Database changes: Create new Flyway migration files (never modify existing ones)
5. **Testing Access**:
• Default user role: ROLE_USER
• Admin/Instructor testing: Update user role in H2 console
• API testing: Use Swagger UI at http://localhost:8080/swagger-ui.html
This knowledge base approach allows Claude Code to maintain context across sessions without re-exploring the entire codebase.