Building a Clean Architecture Banking Application

Spring Boot 3.5 & PostgreSQL

A Production-Ready RESTful Banking API

By: Wildan Anugrah

Github

https://github.com/wildananugrah/spring-learning

By: Wildan Anugrah

Table of Contents

Contents

Note: In HTML export, these links are clickable. In PDF, use page navigation.

Contents (cont.)

💡 **Tip**: Export to HTML for clickable navigation, or use PDF bookmarks feature

What We'll Build

A Fully Functional Banking Application

Application Features

Core Functionality

- User registration and authentication with JWT
- Account creation and management
- Deposit, withdrawal, and transfer operations
- Transaction history with filtering, sorting, and pagination
- Spring Boot Actuator for application monitoring

Architecture

- Clean separation of concerns with three distinct layers
- RESTful API design
- Secure JWT authentication
- PostgreSQL database with HikariCP connection pooling

Architecture Overview

Clean Three-Layered Architecture

The Three Layers

1. Route Layer (`/routes` folder)

Purpose: Handle HTTP requests and responses

Responsibilities:

- Receive incoming requests
- Validate input using Bean Validation
- Send formatted responses
- Handle authentication context

The Three Layers (cont.)

2. Logic Layer (`/logics` folder)

Purpose: Implement business logic

Responsibilities:

- Process business rules
- Coordinate between routes and services
- Handle transactions and data transformations
- Enforce business constraints

The Three Layers (cont.)

3. Service Layer (`/services` folder)

Purpose: Manage dependencies and data access

Responsibilities:

- Database operations (JPA repositories)
- External service integrations
- Configuration management

Architecture Benefits

Why This Separation?

✅ **Maintainability**: Each layer has a single responsibility

✅ **Testability**: Layers can be tested independently

✅ **Scalability**: Easy to modify or replace individual layers

✅ **Readability**: Clear code organization

Project Structure

```
src/main/java/com/user/account/app/
├── config/              # Configuration classes
│   ├── JwtUtil.java
│   ├── JwtAuthenticationFilter.java
│   └── SecurityConfig.java
├── dto/                 # Data Transfer Objects
├── entities/            # JPA Entities
├── exceptions/          # Custom Exceptions
├── logics/              # Business Logic Layer
├── routes/              # Controllers (Route Layer)
└── services/            # Data Access Layer
```

Step 1: Project Setup

Getting Started

Prerequisites

- ☕ Java 17 or higher
- 🔧 Maven 3.6+
- 🐘 PostgreSQL database
- 💻 Your favorite IDE (IntelliJ IDEA, VS Code, etc.)
- 📚 Basic knowledge of Spring Boot and REST APIs

Key Dependencies

```
<!-- Web -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!-- JPA & Database -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>

<!-- Security -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

More Dependencies

```
<!-- Validation -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>

<!-- Actuator for Monitoring -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<!-- Logstash for JSON Logging -->
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>8.0</version>
</dependency>

<!-- Lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
```

JWT Dependencies

```
<!-- JWT Authentication -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.12.3</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.12.3</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.12.3</version>
    <scope>runtime</scope>
</dependency>
```

Maven Compiler Plugin

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.14.1</version>
    <configuration>
        <source>17</source>
        <target>17</target>
        <annotationProcessorPaths>
            <path>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>1.18.36</version>
            </path>
        </annotationProcessorPaths>
    </configuration>
</plugin>
```

Important: This ensures Lombok annotations are processed correctly!

Database Configuration

```
# Application Info
spring.application.name=${APP_NAME:user-account}
server.port=${APP_PORT:8000}

# Database Configuration (using environment variables)
spring.datasource.url=${SPRING_DATASOURCE_URL:jdbc:postgresql://localhost:6435/bank_app}
spring.datasource.username=${SPRING_DATASOURCE_USERNAME:pg}
spring.datasource.password=${SPRING_DATASOURCE_PASSWORD:p@ssw0rd1234}
spring.datasource.driver-class-name=org.postgresql.Driver

# HikariCP Connection Pool
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.idle-timeout=300000
spring.datasource.hikari.max-lifetime=1800000
spring.datasource.hikari.connection-timeout=30000
```

Note:

- Application runs on port **8000** (not 8080)
- Uses environment variables with fallback defaults

JPA Configuration

```
# JPA Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

# JWT Configuration
jwt.secret=YOUR_256_BYTE_SECRET_HERE
jwt.expiration=86400000

# Logging
logging.level.org.springframework.security=DEBUG
```

Generate JWT Secret: `openssl rand -hex 256`

Actuator Configuration

```
# Actuator Configuration
management.endpoints.web.exposure.include=health,info,metrics,env,beans,mappings
management.endpoint.health.show-details=when-authorized
management.health.db.enabled=true
management.info.env.enabled=true

# Application Info
info.app.name=Bank Application API
info.app.description=User Account Management System
info.app.version=1.0.0
```

Step 2: Entity Models

Database Schema Design

User Entity

```java
@Entity
@Table(name = "users")
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false)
    private String password;

    @Column(nullable = false)
    private String fullName;
```

User Entity (cont.)

```
    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private List<Account> accounts;

    @CreationTimestamp
    @Column(nullable = false, updatable = false)
    private LocalDateTime createdAt;

    @UpdateTimestamp
    @Column(nullable = false)
    private LocalDateTime updatedAt;
}
```

Key Features:

- Lombok `@Data`, `@Builder` for clean code
- Automatic timestamp management
- One-to-many relationship with accounts

Account Entity

```
@Entity
@Table(name = "accounts")
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Account {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String accountNumber;

    @Column(nullable = false)
    private String accountName;

    @Column(nullable = false, precision = 19, scale = 2)
    private BigDecimal balance;
```

Account Entity (cont.)

```
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id", nullable = false)
    private User user;

    @OneToMany(mappedBy = "account", cascade = CascadeType.ALL)
    private List<Transaction> transactions;

    @CreationTimestamp
    @Column(nullable = false, updatable = false)
    private LocalDateTime createdAt;

    @UpdateTimestamp
    @Column(nullable = false)
    private LocalDateTime updatedAt;
}
```

Key Features:

- `BigDecimal` for precise currency calculations
- Lazy loading for performance
- Bidirectional relationships

Transaction Entity

```
@Entity
@Table(name = "transactions")
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Transaction {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "account_id", nullable = false)
    private Account account;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    private TransactionType type;

    @Column(nullable = false, precision = 19, scale = 2)
    private BigDecimal amount;
```

Transaction Entity (cont.)

```
    @Column(precision = 19, scale = 2)
    private BigDecimal balanceBefore;

    @Column(precision = 19, scale = 2)
    private BigDecimal balanceAfter;

    @Column
    private String description;

    @Column
    private String referenceNumber;

    @Column
    private String toAccountNumber;

    @Column
    private String fromAccountNumber;

    @CreationTimestamp
    @Column(nullable = false, updatable = false)
    private LocalDateTime createdAt;
```

Transaction Type Enum

```
public enum TransactionType {
    DEPOSIT,
    WITHDRAWAL,
    TRANSFER_IN,
    TRANSFER_OUT
}
```

Key Features:

- Tracks balance before/after each transaction
- Reference numbers for audit trail
- Transfer tracking with account numbers
- Enum for type safety

Step 3: DTOs

Data Transfer Objects

Authentication DTOs

RegisterRequest

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class RegisterRequest {
    @NotBlank(message = "Username is required")
    @Size(min = 3, max = 50)
    private String username;

    @NotBlank(message = "Email is required")
    @Email(message = "Email should be valid")
    private String email;

    @NotBlank(message = "Password is required")
    @Size(min = 6)
    private String password;

    @NotBlank(message = "Full name is required")
    private String fullName;
}
```

Authentication DTOs (cont.)

LoginRequest

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class LoginRequest {
    @NotBlank(message = "Username is required")
    private String username;

    @NotBlank(message = "Password is required")
    private String password;
}
```

Key Features:

- Bean Validation annotations
- Clear validation messages
- Type-safe data transfer

AuthResponse

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class AuthResponse {
    private String token;
    private String type = "Bearer";
    private Long userId;
    private String username;
    private String email;
    private String fullName;
}
```

Returns:

- JWT token for authentication
- User information
- Token type for HTTP header

Generic API Response

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class ApiResponse<T> {
    private boolean success;
    private String message;
    private T data;

    public static <T> ApiResponse<T> success(String message, T data) {
        return ApiResponse.<T>builder()
                .success(true)
                .message(message)
                .data(data)
                .build();
    }

    public static <T> ApiResponse<T> error(String message) {
        return ApiResponse.<T>builder()
                .success(false)
                .message(message)
                .build();
    }
}
```

Why Use DTOs?

Benefits

✅ **Separation of Concerns**: API contracts separate from entities

✅ **Security**: Don't expose internal entity structure

✅ **Validation**: Input validation at API boundary

✅ **Flexibility**: Change entities without breaking API

✅ **Performance**: Transfer only needed data

Step 4: Exception Handling

Clean Error Management

Custom Exceptions

```
// ResourceNotFoundException.java
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}

// DuplicateResourceException.java
public class DuplicateResourceException extends RuntimeException {
    public DuplicateResourceException(String message) {
        super(message);
    }
}

// InsufficientBalanceException.java
public class InsufficientBalanceException extends RuntimeException {
    public InsufficientBalanceException(String message) {
        super(message);
    }
}
```

Global Exception Handler

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ApiResponse<Void>> handleResourceNotFound(
            ResourceNotFoundException ex) {
        return ResponseEntity
                .status(HttpStatus.NOT_FOUND)
                .body(ApiResponse.error(ex.getMessage()));
    }

    @ExceptionHandler(DuplicateResourceException.class)
    public ResponseEntity<ApiResponse<Void>> handleDuplicateResource(
            DuplicateResourceException ex) {
        return ResponseEntity
                .status(HttpStatus.CONFLICT)
                .body(ApiResponse.error(ex.getMessage()));
    }
```

Validation Exception Handler

```
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ApiResponse<Map<String, String>>>
            handleValidation(MethodArgumentNotValidException ex) {

        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getAllErrors().forEach((error) -> {
            String fieldName = ((FieldError) error).getField();
            String errorMessage = error.getDefaultMessage();
            errors.put(fieldName, errorMessage);
        });

        return ResponseEntity
                .status(HttpStatus.BAD_REQUEST)
                .body(ApiResponse.<Map<String, String>>builder()
                        .success(false)
                        .message("Validation failed")
                        .data(errors)
                        .build());
    }
}
```

Benefits of @RestControllerAdvice

Why Use It?

✅ **Centralized**: All exception handling in one place

✅ **Clean Controllers**: No try-catch blocks needed

✅ **Consistent**: Same error response format everywhere

✅ **Maintainable**: Easy to add new exception types

✅ **Professional**: Proper HTTP status codes

Step 5: Repository Layer

Data Access with JPA

UserRepository

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);

    Optional<User> findByEmail(String email);

    boolean existsByUsername(String username);

    boolean existsByEmail(String email);
}
```

Features:

- Spring Data JPA - no implementation needed!
- Type-safe queries
- Automatic CRUD operations

AccountRepository

```
@Repository
public interface AccountRepository extends JpaRepository<Account, Long> {
    Optional<Account> findByAccountNumber(String accountNumber);

    List<Account> findByUserId(Long userId);

    boolean existsByAccountNumber(String accountNumber);
}
```

Features:

- Custom query methods
- Derived queries from method names
- Spring generates implementation automatically

TransactionRepository

```
@Repository
public interface TransactionRepository
        extends JpaRepository<Transaction, Long> {

    Page<Transaction> findByAccountId(Long accountId, Pageable pageable);

    @Query("SELECT t FROM Transaction t WHERE t.account.id = :accountId " +
           "AND (CAST(:startDate AS timestamp) IS NULL " +
           "OR t.createdAt >= :startDate) " +
           "AND (CAST(:endDate AS timestamp) IS NULL " +
           "OR t.createdAt <= :endDate)")
    Page<Transaction> findByAccountIdAndDateRange(
            @Param("accountId") Long accountId,
            @Param("startDate") LocalDateTime startDate,
            @Param("endDate") LocalDateTime endDate,
            Pageable pageable
    );
}
```

Important: CAST for Nullable Parameters

Why CAST is Necessary

```
// ❌ This FAILS with PostgreSQL
"WHERE :startDate IS NULL OR t.createdAt >= :startDate"

// ✅ This WORKS
"WHERE CAST(:startDate AS timestamp) IS NULL OR t.createdAt >= :startDate"
```

Reason: PostgreSQL cannot determine the data type of nullable parameters in JPQL queries without explicit casting.

Error Without CAST: `could not determine data type of parameter $2`

Step 6: JWT Authentication

Secure Token-Based Auth

JWT Utility Class

```
@Component
public class JwtUtil {
    @Value("${jwt.secret}")
    private String secret;

    @Value("${jwt.expiration}")
    private Long expiration;

    private SecretKey getSigningKey() {
        return Keys.hmacShaKeyFor(secret.getBytes(StandardCharsets.UTF_8));
    }

    @SneakyThrows
    public String generateToken(String username, Long userId) {
        Map<String, Object> claims = new HashMap<>();
        claims.put("userId", userId);
        return Jwts.builder()
                .claims(claims)
                .subject(username)
```

JWT Utility Class (cont.)

```
                .issuedAt(new Date(System.currentTimeMillis()))
                .expiration(new Date(System.currentTimeMillis() + expiration))
                .signWith(getSigningKey())
                .compact();
    }

    @SneakyThrows
    public String extractUsername(String token) {
        return Jwts.parser()
                .verifyWith(getSigningKey())
                .build()
                .parseSignedClaims(token)
                .getPayload()
                .getSubject();
    }
```

Note: `@SneakyThrows` from Lombok keeps code clean!

JWT Validation

```
    @SneakyThrows
    public Boolean validateToken(String token, String username) {
        final String extractedUsername = extractUsername(token);
        Date expiration = Jwts.parser()
                .verifyWith(getSigningKey())
                .build()
                .parseSignedClaims(token)
                .getPayload()
                .getExpiration();

        return (extractedUsername.equals(username) &&
                !expiration.before(new Date()));
    }
}
```

Validates:

- Token signature
- Username matches
- Token not expired

JWT Authentication Filter

```
@Component
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {
    private final JwtUtil jwtUtil;

    @Override
    @SneakyThrows
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) {
        final String authHeader = request.getHeader("Authorization");

        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }

        final String jwt = authHeader.substring(7);
        final String username = jwtUtil.extractUsername(jwt);
```

JWT Authentication Filter (cont.)

```
        if (username != null &&
            SecurityContextHolder.getContext().getAuthentication() == null) {

            if (jwtUtil.validateToken(jwt, username)) {
                UsernamePasswordAuthenticationToken authToken =
                    new UsernamePasswordAuthenticationToken(
                        username, null, new ArrayList<>()
                    );
                authToken.setDetails(
                    new WebAuthenticationDetailsSource().buildDetails(request)
                );
                SecurityContextHolder.getContext()
                    .setAuthentication(authToken);
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

Security Configuration

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig {
    private final JwtAuthenticationFilter jwtAuthenticationFilter;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
            throws Exception {
        http
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/actuator/**").permitAll()
                .anyRequest().authenticated()
            )
```

Security Configuration (cont.)

```
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )
            .addFilterBefore(jwtAuthenticationFilter,
                             UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

Key Points:

- Stateless sessions (JWT-based)
- Public auth endpoints
- BCrypt password hashing

Step 7: Logic Layer

Business Logic Implementation

TransactionLogic - Deposit

```
@Service
@RequiredArgsConstructor
public class TransactionLogic {
    private final TransactionRepository transactionRepository;
    private final AccountRepository accountRepository;
    private final UserRepository userRepository;

    @Transactional
    @SneakyThrows
    public TransactionResponse deposit(String username,
                                       TransactionRequest request) {
        validateAccountOwnership(username, request.getAccountNumber());

        Account account = accountRepository
            .findByAccountNumber(request.getAccountNumber())
            .orElseThrow(() ->
                new ResourceNotFoundException("Account not found"));
```

TransactionLogic - Deposit (cont.)

```
        BigDecimal balanceBefore = account.getBalance();
        BigDecimal balanceAfter = balanceBefore.add(request.getAmount());

        account.setBalance(balanceAfter);
        accountRepository.save(account);

        Transaction transaction = Transaction.builder()
                .account(account)
                .type(Transaction.TransactionType.DEPOSIT)
                .amount(request.getAmount())
                .balanceBefore(balanceBefore)
                .balanceAfter(balanceAfter)
                .description(request.getDescription())
                .referenceNumber(generateReferenceNumber())
                .build();

        transaction = transactionRepository.save(transaction);
        return mapToResponse(transaction);
    }
```

Account Ownership Validation

```
    @SneakyThrows
    private void validateAccountOwnership(String username,
                                          String accountNumber) {
        User user = userRepository.findByUsername(username)
                .orElseThrow(() ->
                    new ResourceNotFoundException("User not found"));

        Account account = accountRepository
                .findByAccountNumber(accountNumber)
                .orElseThrow(() ->
                    new ResourceNotFoundException("Account not found"));

        if (!account.getUser().getId().equals(user.getId())) {
            throw new ResourceNotFoundException("Account not found");
        }
    }
```

Security: Users can only access their own accounts!

Reference Number Generation

```
    private String generateReferenceNumber() {
        return "TXN" + UUID.randomUUID()
                .toString()
                .replace("-", "")
                .substring(0, 16)
                .toUpperCase();
    }
```

Example: `TXN1A2B3C4D5E6F7G8H`

Purpose: Unique identifier for audit trail

Key Logic Layer Concepts

Important Annotations

@Transactional: Ensures atomicity

- All operations succeed or all fail
- Automatic rollback on exceptions
- Database consistency guaranteed

@SneakyThrows: Clean code

- Avoids cluttering with try-catch
- Lombok annotation
- Better readability

Step 8: Route Layer

Controllers & API Endpoints

TransactionController

```
@RestController
@RequestMapping("/api/transactions")
@RequiredArgsConstructor
public class TransactionController {
    private final TransactionLogic transactionLogic;

    @PostMapping("/deposit")
    public ResponseEntity<ApiResponse<TransactionResponse>> deposit(
            Authentication authentication,
            @Valid @RequestBody TransactionRequest request) {

        TransactionResponse response = transactionLogic.deposit(
            authentication.getName(),
            request
        );

        return ResponseEntity
                .status(HttpStatus.CREATED)
                .body(ApiResponse.success("Deposit successful", response));
    }
```

Transaction History Endpoint

```
    @GetMapping
    public ResponseEntity<ApiResponse<Page<TransactionResponse>>>
            getTransactionHistory(
                Authentication authentication,
                @RequestParam String accountNumber,
                @RequestParam(required = false) LocalDateTime startDate,
                @RequestParam(required = false) LocalDateTime endDate,
                @RequestParam(defaultValue = "0") int page,
                @RequestParam(defaultValue = "10") int size,
                @RequestParam(defaultValue = "createdAt") String sortBy,
                @RequestParam(defaultValue = "desc") String sortDirection) {

        Page<TransactionResponse> response =
            transactionLogic.getTransactionHistory(
                authentication.getName(), accountNumber,
                startDate, endDate, page, size, sortBy, sortDirection
            );
```

Transaction History Response

```
        return ResponseEntity.ok(
            ApiResponse.success("Transaction history retrieved", response)
        );
    }
}
```

Features:

- Pagination support
- Date range filtering
- Sorting capabilities
- Authentication required

Controller Responsibilities

What Controllers Should Do

✅ Receive HTTP requests

✅ Extract authentication information

✅ Validate input with `@Valid`

✅ Call business logic layer

✅ Return formatted responses

What Controllers Should NOT Do

❌ Business logic

❌ Database operations

❌ Complex calculations

Step 9: Docker Deployment

Containerization with Docker

Why Docker?

Benefits of Containerization

✅ **Consistency**: Same environment everywhere

✅ **Portability**: Run anywhere Docker is installed

✅ **Isolation**: Dependencies contained

✅ **Scalability**: Easy to scale with orchestration

✅ **Simplicity**: One command to run

Dockerfile - Multi-Stage Build

Stage 1: Build

```
# Stage 1: Build the application
FROM maven:3.9.6-eclipse-temurin-17 AS build

WORKDIR /app

# Copy pom.xml and download dependencies (cached layer)
COPY pom.xml .
RUN mvn dependency:go-offline -B

# Copy source code
COPY src ./src

# Build the application (skip tests for faster builds)
RUN mvn clean package -DskipTests
```

Dockerfile - Multi-Stage Build (cont.)

Stage 2: Run

```
# Stage 2: Run the application
FROM eclipse-temurin:17-jre

WORKDIR /app

# Create a non-root user for security
RUN groupadd -r spring && useradd -r -g spring spring

# Copy the JAR from build stage
COPY --from=build /app/target/*.jar app.jar

# Change ownership to spring user
RUN chown -R spring:spring /app

# Switch to non-root user
USER spring:spring

# Expose port 8000
EXPOSE 8000
```

Dockerfile - Health Check & Entrypoint

```
# Health check (using curl)
HEALTHCHECK --interval=30s --timeout=3s --start-period=60s --retries=3 \
  CMD curl -f http://localhost:8000/actuator/health || exit 1

# JVM configuration for containerized environment
ENV JAVA_OPTS="-XX:+UseContainerSupport -XX:MaxRAMPercentage=75.0 -XX:InitialRAMPercentage=50.0"

# Run the application
ENTRYPOINT ["sh", "-c", "java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar app.jar"]
```

Key Features:

- Multi-stage build (smaller image)
- Non-root user (security)
- Health check (monitoring)
- JVM optimization for containers

Docker Compose Configuration

```yaml
services:
  # Spring Boot Application
  app:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: bank-app
    restart: unless-stopped
    env_file:
      - .env
    environment:
      APP_PORT: 8000
    ports:
      - "8000:8000"
    networks:
      app-net: {}
    healthcheck:
      test: ["CMD", "wget", "--no-verbose", "--tries=1", "--spider", "http://localhost:8000/actuator/health"]
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 60s
```

Docker Compose (cont.)

```
networks:
  app-net:
    external: true
    name: "user-acccount-net"

volumes:
  postgres_data:
    driver: local
```

Key Features:

- Environment variable support (`.env` file)
- Health checks
- Automatic restart
- Network isolation
- Volume persistence

Docker Commands

Build and Run

```
# Build and start the application
docker compose up --build -d

# Check container status
docker compose ps

# View logs
docker compose logs -f app

# Stop the application
docker compose down

# Remove volumes (clean start)
docker compose down -v
```

Docker Best Practices

Security

✅ **Non-root user**: Run as `spring:spring` user

✅ **Multi-stage build**: Smaller attack surface

✅ **No secrets in image**: Use environment variables

Performance

✅ **Layer caching**: Dependencies downloaded separately

✅ **JVM optimization**: Container-aware memory settings

✅ **Health checks**: Automatic restart on failure

Monitoring

✅ **Actuator integration**: Health check endpoint

✅ **Log aggregation**: `docker compose logs`

Step 10: Testing the Application

Running & Testing

Setup Database

```
-- Create database
CREATE DATABASE bank_app;

-- PostgreSQL should be running on port 6435
-- (or adjust in application.properties)
```

Environment Variables Setup

Create a `.env` file in the project root:

```
# .env file
SPRING_DATASOURCE_URL=jdbc:postgresql://localhost:6435/bank_app
SPRING_DATASOURCE_USERNAME=pg
SPRING_DATASOURCE_PASSWORD=p@ssw0rd1234
APP_PORT=8000
JWT_SECRET=your_256_byte_secret_here
JWT_EXPIRATION=86400000
```

Generate JWT Secret: `openssl rand -hex 256`

Run the Application

Option 1: Using Maven with Environment Variables

```
# Load environment variables and run
export $(cat .env | xargs) && ./mvnw spring-boot:run

# Or use the provided script
./run.sh
```

Option 2: Using Docker Compose

```
# Build and run with Docker
docker compose up --build -d

# Check status
docker compose ps

# View logs
docker compose logs -f app

# Stop the application
docker compose down
```

Application starts on: `http://localhost:8000`

Test Endpoints - Register

```
### Register a new user
POST http://localhost:8000/api/auth/register
Content-Type: application/json

{
  "username": "john_doe",
  "email": "john@example.com",
  "password": "password123",
  "fullName": "John Doe"
}
```

Response:

```
{
  "success": true,
  "message": "User registered successfully",
  "data": {
    "token": "eyJhbGciOiJIUzUxMiJ9...",
    "type": "Bearer",
    "userId": 1,
    "username": "john_doe"
  }
}
```

Test Endpoints - Login

```
### Login
POST http://localhost:8000/api/auth/login
Content-Type: application/json

{
  "username": "john_doe",
  "password": "password123"
}
```

Save the token from response for subsequent requests!

Test Endpoints - Create Account

```
### Create account
POST http://localhost:8000/api/accounts
Content-Type: application/json
Authorization: Bearer YOUR_TOKEN_HERE

{
  "accountName": "Savings Account",
  "initialBalance": 1000.00
}
```

Response:

```
{
  "success": true,
  "message": "Account created successfully",
  "data": {
    "accountNumber": "ACC1234567890",
    "accountName": "Savings Account",
    "balance": 1000.00
  }
}
```

Test Endpoints - Deposit

```
### Deposit money
POST http://localhost:8000/api/transactions/deposit
Content-Type: application/json
Authorization: Bearer YOUR_TOKEN_HERE

{
  "accountNumber": "ACC1234567890",
  "amount": 500.00,
  "description": "Salary deposit"
}
```

Test Endpoints - Transfer

```
### Transfer money
POST http://localhost:8000/api/transactions/transfer
Content-Type: application/json
Authorization: Bearer YOUR_TOKEN_HERE

{
  "fromAccountNumber": "ACC1234567890",
  "toAccountNumber": "ACC9876543210",
  "amount": 200.00,
  "description": "Payment to friend"
}
```

Test Endpoints - Transaction History

```
### Get transaction history with filters
GET http://localhost:8000/api/transactions?accountNumber=ACC1234567890&page=0&size=10&sortBy=createdAt&sortDirection=desc
Authorization: Bearer YOUR_TOKEN_HERE

### With date range filter
GET http://localhost:8000/api/transactions?accountNumber=ACC1234567890&startDate=2025-01-01T00:00:00&endDate=2025-12-31T23:59:59
Authorization: Bearer YOUR_TOKEN_HERE
```

Step 11: Spring Boot Actuator

Production-Ready Monitoring

What is Actuator?

Production-Ready Features

Spring Boot Actuator provides:

- 📊 **Health checks**
- 📈 **Metrics collection**
- 📋 **Application information**
- 🔍 **Environment details**
- 🌐 **HTTP trace**
- 🔧 **Configuration properties**

Purpose: Monitor and manage your application in production

Available Actuator Endpoints

```
### Health Check
GET http://localhost:8000/actuator/health

### Application Info
GET http://localhost:8000/actuator/info

### All Available Metrics
GET http://localhost:8000/actuator/metrics

### Specific Metrics
GET http://localhost:8000/actuator/metrics/http.server.requests
GET http://localhost:8000/actuator/metrics/jvm.memory.used
GET http://localhost:8000/actuator/metrics/system.cpu.usage
```

Database Connection Metrics

```
### HikariCP Connection Pool Metrics
GET http://localhost:8000/actuator/metrics/hikaricp.connections.active
GET http://localhost:8000/actuator/metrics/hikaricp.connections.idle
GET http://localhost:8000/actuator/metrics/hikaricp.connections.max
GET http://localhost:8000/actuator/metrics/hikaricp.connections.min
```

Why Monitor Connections?

- Track connection pool usage
- Identify connection leaks
- Optimize pool configuration
- Prevent database bottlenecks

Environment & Configuration

```
### Environment Variables and Properties
GET http://localhost:8000/actuator/env

### Application Beans
GET http://localhost:8000/actuator/beans

### Request Mappings (All Endpoints)
GET http://localhost:8000/actuator/mappings
```

Health Check Response Example

```
{
  "status": "UP",
  "components": {
    "db": {
      "status": "UP",
      "details": {
        "database": "PostgreSQL",
        "validationQuery": "isValid()"
      }
    },
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 500068036608,
        "free": 198648901632,
        "threshold": 10485760
      }
    }
  }
}
```

Metrics Response Example

```
{
  "name": "http.server.requests",
  "measurements": [
    {
      "statistic": "COUNT",
      "value": 157.0
    },
    {
      "statistic": "TOTAL_TIME",
      "value": 3.456789
    }
  ],
  "availableTags": [
    {
      "tag": "method",
      "values": ["GET", "POST", "PUT", "DELETE"]
    }
  ]
}
```

Production Considerations

Security Best Practices

1. **Secure Actuator Endpoints**

```
.requestMatchers("/actuator/**").hasRole("ADMIN")
```

2. **Limit Exposed Endpoints**

```
management.endpoints.web.exposure.include=health,info,metrics
```

3. **Hide Sensitive Information**

```
management.endpoint.env.show-values=WHEN_AUTHORIZED
```

Production Considerations (cont.)

Integration with Monitoring Tools

Prometheus Integration:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Grafana Dashboard: Visualize metrics

ELK Stack: Log aggregation and analysis

CloudWatch: AWS monitoring

Step 12: Comprehensive Logging

Request/Response Logging with JSON Support

What is Logging?

Why Comprehensive Logging Matters

In production environments, you need to know:

- 🔍 **What happened?** - Every request and response
- 👤 **Who did it?** - User identification
- ⏱️ **How long?** - Performance tracking
- ❌ **What went wrong?** - Error details with context

Our logging system provides all of this automatically!

Logging Features

Automatic Request/Response Logging

✅ **Timestamp**: Millisecond precision (yyyy-MM-dd HH:mm:ss.SSS)

✅ **User ID**: Extracted from JWT authentication

✅ **HTTP Method & URI**: Full request path with query params

✅ **Status Code**: HTTP response status

✅ **Elapsed Time**: Request processing time in milliseconds

✅ **Request/Response Body**: For errors only (with sensitive data masked)

Dual Format Support

Two Logging Formats for Different Needs

Human-Readable (Development):

```
2025-11-18 21:30:45.123 INFO - Timestamp: 2025-11-18 21:30:45.123 | User ID: john@example.com | Method: GET | URI: /api/accounts | Status: 200 | Elapsed Time: 45 ms
```

JSON Format (Production):

```
{
  "timestamp": "2025-11-18T14:30:45.123Z",
  "level": "INFO",
  "userId": "john@example.com",
  "httpMethod": "GET",
  "httpUri": "/api/accounts",
  "httpStatusCode": "200",
  "elapsedTime": "45"
}
```

Switching Between Formats

Easy Profile-Based Configuration

Development (human-readable):

```
./mvnw spring-boot:run
```

Production (JSON):

```
./mvnw spring-boot:run -Dspring-boot.run.profiles=json
```

Docker:

```
docker run -e SPRING_PROFILES_ACTIVE=json bank-app
```

Error Logging Example

Detailed Error Information

```
==================== ERROR REQUEST/RESPONSE ====================
Timestamp: 2025-11-18 21:31:15.456
User ID: john@example.com
Method: POST
URI: /api/accounts/123/withdraw
Status Code: 400
Elapsed Time: 12 ms
--- Request Body ---
{"amount":5000.00}
--- Response Body ---
{"success":false,"message":"Insufficient balance","data":null}
================================================================
```

Note: Only errors show request/response bodies. Success requests show summary only.

Security Features

Sensitive Data Masking

Automatically masks sensitive information:

Before Masking:

```
{"email":"user@example.com","password":"mySecret123"}
```

After Masking (in logs):

```
{"email":"user@example.com","password":"***MASKED***"}
```

Fields Masked: `password`, `token`, `secret`

Implementation Components

RequestResponseLoggingFilter

```
@Slf4j
@Component
@Order(Ordered.HIGHEST_PRECEDENCE)
public class RequestResponseLoggingFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) {
        long startTime = System.currentTimeMillis();

        ContentCachingRequestWrapper wrappedRequest =
            new ContentCachingRequestWrapper(request);
        ContentCachingResponseWrapper wrappedResponse =
            new ContentCachingResponseWrapper(response);

        try {
            filterChain.doFilter(wrappedRequest, wrappedResponse);
        } finally {
            long elapsedTime = System.currentTimeMillis() - startTime;
            logRequestResponse(wrappedRequest, wrappedResponse, elapsedTime);
            wrappedResponse.copyBodyToResponse();
        }
    }
}
```

Logback Configuration

logback-spring.xml

```
<configuration>
    <!-- Human-readable appender -->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %highlight(%-5level)
                     [%thread] %cyan(%logger{36}) - %msg%n</pattern>
        </encoder>
    </appender>

    <!-- JSON appender -->
    <appender name="CONSOLE_JSON"
              class="ch.qos.logback.core.ConsoleAppender">
        <encoder class="net.logstash.logback.encoder.LogstashEncoder">
            <includeMdcKeyName>userId</includeMdcKeyName>
            <includeMdcKeyName>httpMethod</includeMdcKeyName>
            <includeMdcKeyName>httpUri</includeMdcKeyName>
            <includeMdcKeyName>httpStatusCode</includeMdcKeyName>
            <includeMdcKeyName>elapsedTime</includeMdcKeyName>
        </encoder>
    </appender>
```

Logback Configuration (cont.)

```
    <!-- Profile: default (human-readable) -->
    <springProfile name="default,dev,local">
        <root level="INFO">
            <appender-ref ref="CONSOLE"/>
        </root>
    </springProfile>

    <!-- Profile: json (JSON for production) -->
    <springProfile name="json,prod,production">
        <root level="INFO">
            <appender-ref ref="CONSOLE_JSON"/>
        </root>
    </springProfile>
</configuration>
```

Key: Use Spring profiles to switch between formats!

Log Aggregation Integration

Ready for Production Monitoring

ELK Stack (Elasticsearch, Logstash, Kibana):

```
input {
  file {
    path => "/app/logs/application.json"
    codec => "json"
  }
}
output {
  elasticsearch {
    hosts => ["localhost:9200"]
    index => "bank-app-%{+YYYY.MM.dd}"
  }
}
```

Also Compatible With:

- AWS CloudWatch
- Splunk
- Datadog
- Grafana Loki

Querying JSON Logs

Using jq for Analysis

```
# Get all error logs
cat logs/application.json | jq 'select(.level == "ERROR")'

# Get logs for specific user
cat logs/application.json | jq 'select(.userId == "john@example.com")'

# Get slow requests (> 1000ms)
cat logs/application.json | jq 'select(.elapsedTime | tonumber > 1000)'

# Count requests per endpoint
cat logs/application.json | jq -r '.httpUri' | sort | uniq -c

# Calculate average response time
cat logs/application.json | jq -r '.elapsedTime' | \
  awk '{sum+=$1; n++} END {print sum/n}'
```

Logging Benefits

Why This Matters in Production

✅ **Debugging**: Find exactly what went wrong and why

✅ **Performance Monitoring**: Identify slow endpoints

✅ **Security Auditing**: Track who accessed what

✅ **Business Analytics**: API usage patterns

✅ **Compliance**: Audit trail for regulations

✅ **Alerting**: Trigger alerts on errors or slow requests

Documentation

Comprehensive Guides Available

📚 **LOGGING_README.md**: Quick start guide

📖 **LOGGING_GUIDE.md**: Human-readable logging

📘 **JSON_LOGGING_GUIDE.md**: JSON logging & integration

🧪 **TEST_LOGGING.md**: Testing the logging system

📊 **LOGGING_EXAMPLES.md**: Format comparisons

Key Features Implemented

What We've Built

Feature 1: Clean Architecture

Three-Layered Design

✅ **Route Layer**: HTTP handling & validation

✅ **Logic Layer**: Business rules & transactions

✅ **Service Layer**: Data access & repositories

Benefits:

- Clear separation of concerns
- Easy to test independently
- Maintainable and scalable
- Professional code organization

Feature 2: JWT Authentication

Secure Token-Based Auth

✅ Stateless sessions (no server-side storage)

✅ Secure token generation with HMAC-SHA

✅ Token validation on every request

✅ Protected endpoints with Spring Security

Benefits:

- Scalable (no session storage)
- Secure (encrypted tokens)
- RESTful (stateless)
- Industry standard

Feature 3: Global Exception Handling

@RestControllerAdvice

✅ Centralized error management

✅ Consistent error responses

✅ Proper HTTP status codes

✅ Validation error handling

Benefits:

- Clean controller code
- Professional API responses
- Easy to maintain
- Better debugging

Feature 4: SneakyThrows Usage

Lombok's @SneakyThrows

```
// Without @SneakyThrows
public void deposit() {
    try {
        // logic
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

// With @SneakyThrows
@SneakyThrows
public void deposit() {
    // logic
}
```

Benefits: Cleaner code, better readability

Feature 5: Transaction Management

@Transactional

✅ **Atomicity**: All or nothing

✅ **Consistency**: Data integrity maintained

✅ **Isolation**: Concurrent transaction handling

✅ **Durability**: Permanent once committed

Example: Transfer operation - both debit and credit succeed, or both fail!

Feature 6: Pagination & Filtering

Efficient Data Retrieval

✅ Page-based results (avoid loading all data)

✅ Custom page size

✅ Date range filtering

✅ Sorting capabilities (ASC/DESC)

Example:

```
GET /api/transactions?page=0&size=10&sortBy=createdAt&sortDirection=desc
```

Feature 7: Spring Boot Actuator

Production Monitoring

✅ Health checks

✅ Performance metrics

✅ Database connection monitoring

✅ Application insights

Benefits: Know what's happening in production!

Feature 8: HikariCP Connection Pooling

High-Performance Database Connections

✅ Connection pooling (reuse connections)

✅ Configurable pool size

✅ Connection timeout handling

✅ Leak detection

Configuration:

- Min idle: 5
- Max pool size: 20
- Connection timeout: 30s
- Idle timeout: 5m

Feature 8: Comprehensive Logging System

Production-Ready Logging

✅ Dual format support (human-readable + JSON)

✅ Automatic request/response logging

✅ Sensitive data masking

✅ Performance tracking (elapsed time)

✅ User identification from JWT

✅ Error details with request/response bodies

Benefits: Debug issues, monitor performance, audit access, integrate with log aggregation systems

Best Practices Demonstrated

Professional Spring Boot Development

Best Practice 1: DTOs for Data Transfer

Separate Entities from API Contracts

✅ **Security**: Don't expose internal structure

✅ **Flexibility**: Change entities without breaking API

✅ **Validation**: Validate at API boundary

✅ **Documentation**: Clear API contracts

Example: UserEntity has password, but UserResponse does not!

Best Practice 2: Bean Validation

Automatic Input Validation

```
@NotBlank(message = "Username is required")
@Size(min = 3, max = 50)
private String username;

@Email(message = "Email should be valid")
private String email;
```

✅ Declarative validation

✅ Consistent error messages

✅ No boilerplate validation code

Best Practice 3: Builder Pattern

Clean Object Creation with Lombok

```
User user = User.builder()
    .username("john_doe")
    .email("john@example.com")
    .fullName("John Doe")
    .build();
```

✅ Readable code

✅ Immutable objects

✅ Optional parameters

✅ No constructor pollution

Best Practice 4: Repository Pattern

Abstract Data Access

```
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
}
```

✅ No SQL code needed

✅ Type-safe queries

✅ Easy to test (mockable)

✅ Switch databases easily

Best Practice 5: Dependency Injection

Loose Coupling with Constructor Injection

```
@RequiredArgsConstructor
public class TransactionLogic {
    private final TransactionRepository transactionRepository;
    private final AccountRepository accountRepository;
}
```

✅ **Immutable** (final fields)

✅ **Testable** (inject mocks)

✅ **Explicit** dependencies

✅ **Thread-safe**

Best Practice 6: RESTful Design

Standard HTTP Methods & Status Codes

Operation	Method	Status Code
Create	POST	201 Created
Read	GET	200 OK
Update	PUT/PATCH	200 OK
Delete	DELETE	204 No Content
Error	Any	400/404/500

✅ Predictable API

✅ Industry standard

✅ Easy to understand

Best Practice 7: Security

Multiple Security Layers

✅ **JWT**: Stateless authentication

✅ **BCrypt**: Password hashing

✅ **Ownership Validation**: Users access only their data

✅ **Input Validation**: Prevent injection attacks

✅ **HTTPS**: Encrypt data in transit (production)

Project Structure Recap

Clean Organization

Complete Project Structure

```
src/main/java/com/user/account/app/
├── config/
│   ├── JwtUtil.java
│   ├── JwtAuthenticationFilter.java
│   ├── SecurityConfig.java
│   ├── RequestResponseLoggingFilter.java
│   ├── CachedBodyHttpServletRequest.java
│   └── CachedBodyHttpServletResponse.java
├── dto/
│   ├── RegisterRequest.java
│   ├── LoginRequest.java
│   ├── AuthResponse.java
│   ├── AccountRequest.java
│   ├── AccountResponse.java
│   ├── TransactionRequest.java
│   ├── TransactionResponse.java
│   └── ApiResponse.java
```

Project Structure (cont.)

```
├── entities/
│   ├── User.java
│   ├── Account.java
│   └── Transaction.java
├── exceptions/
│   ├── ResourceNotFoundException.java
│   ├── DuplicateResourceException.java
│   ├── InsufficientBalanceException.java
│   └── GlobalExceptionHandler.java
├── logics/
│   ├── AuthLogic.java
│   ├── AccountLogic.java
│   └── TransactionLogic.java
```

Project Structure (cont.)

```
├── routes/
│   ├── AuthController.java
│   ├── AccountController.java
│   └── TransactionController.java
├── services/
│   ├── UserRepository.java
│   ├── AccountRepository.java
│   └── TransactionRepository.java
└── UserAccountApplication.java
```

Conclusion

What We've Accomplished

Summary: What We Built

A production-ready banking application with:

- ✅ Clean three-layered architecture
- ✅ Secure JWT authentication
- ✅ Comprehensive exception handling
- ✅ Transaction management with ACID guarantees
- ✅ Pagination and filtering capabilities
- ✅ Spring Boot Actuator monitoring
- ✅ HikariCP connection pooling
- ✅ PostgreSQL with optimized queries
- ✅ RESTful API design
- ✅ Docker containerization with multi-stage builds
- ✅ Environment variable configuration
- ✅ **Comprehensive logging system (human-readable + JSON)**
- ✅ **Log aggregation ready (ELK, Splunk, CloudWatch)**

Why This Architecture?

The Benefits

Maintainability: Easy to locate and modify code

Testability: Each layer tested independently

Scalability: Easy to add new features

Readability: Clear organization and purpose

Professional: Industry best practices

Next Steps

Extending the Application

Suggested Enhancements

Feature Additions

1. **User Roles & Permissions**: Admin, Manager, Customer
2. **Account Statements**: PDF generation
3. **Email Notifications**: Transaction alerts
4. **Transaction Limits**: Daily/monthly limits
5. ~~**Audit Logging**~~ ✅ **Implemented with logging system!**
6. **API Rate Limiting**: Prevent abuse
7. **Swagger/OpenAPI**: API documentation

Additional Improvements

Technical Enhancements

1. **Redis Caching**: Improve performance
2. **Message Queue**: Async processing (RabbitMQ/Kafka)
3. ~~**Docker**: Containerization~~ ✅ **Implemented!**
4. **Kubernetes**: Orchestration
5. **CI/CD Pipeline**: Automated deployment
6. **Integration Tests**: Full API testing
7. **Load Testing**: Performance verification

Resources

Documentation

Project Files

📚 **API_DOCUMENTATION.md**: Detailed API specifications

🧪 **api-tests.http**: Testing examples

📖 **MEDIUM_ARTICLE.md**: This tutorial

🔧 **application.properties**: Configuration reference

📋 **LOGGING_README.md**: Logging quick start

📘 **JSON_LOGGING_GUIDE.md**: JSON logging integration

Testing the Application

Quick Start Commands

Option 1: Maven

```
# Build
./mvnw clean install

# Run with environment variables
export $(cat .env | xargs) && ./mvnw spring-boot:run

# Or use the script
./run.sh
```

Option 2: Docker

```
# Build and run
docker compose up --build -d

# View logs
docker compose logs -f app
```

Test the application:

```
curl http://localhost:8000/actuator/health
```

Base URL: `http://localhost:8000` | **Note**: Port 8000 (not 8080)!

Bonus: N+1 Query Problem

Common Performance Issue & Solutions

What is the N+1 Query Problem?

The Problem

The N+1 query problem is a common performance issue in JPA/Hibernate where:

- **1 query** fetches N parent entities
- **N queries** fetch related child entities (one query per parent)

This results in **1 + N = N+1 queries** instead of a single optimized query.

Example Entities

- `Author` (parent) - has many books
- `Book` (child) - belongs to one author

The Problem in Action

```
List<Author> authors = authorRepository.findAll(); // 1 query
for (Author author : authors) {
    author.getBooks().size(); // N queries (one per author!)
}
```

SQL Executed:

```
-- Query 1: Fetch all authors
SELECT * FROM authors;

-- Query 2: For author 1
SELECT * FROM books WHERE author_id = 1;

-- Query 3: For author 2
SELECT * FROM books WHERE author_id = 2;

-- ... and so on for each author
```

With 5 authors: 1 + 5 = **6 queries**
With 100 authors: 1 + 100 = **101 queries!** ❌

How to Identify N+1 Problems

Enable SQL Logging

```
# application.properties
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.generate_statistics=true
```

Watch Console Output

You'll see multiple SELECT queries executing:

```
Hibernate: select a1_0.id,a1_0.email,a1_0.name from authors a1_0
Hibernate: select b1_0.author_id,b1_0.id,b1_0.isbn,b1_0.title
           from books b1_0 where b1_0.author_id=?
Hibernate: select b1_0.author_id,b1_0.id,b1_0.isbn,b1_0.title
           from books b1_0 where b1_0.author_id=?
...
```

Solution 1: FETCH JOIN (Recommended)

Use JPQL with JOIN FETCH

```
@Query("SELECT DISTINCT a FROM Author a LEFT JOIN FETCH a.books")
List<Author> findAllWithBooks();
```

SQL Executed:

```
SELECT DISTINCT a.*, b.*
FROM authors a
LEFT JOIN books b ON a.id = b.author_id;
```

Result: Only 1 query! ✅

When to use:

- When you ALWAYS need the associated data
- For read-heavy operations
- When the association size is reasonable

Solution 2: @EntityGraph

Dynamic Fetching Strategy

```
@Query("SELECT a FROM Author a")
@EntityGraph(attributePaths = {"books"})
List<Author> findAllWithBooksEntityGraph();
```

Result: Only 1 query! ✅

When to use:

- More flexible than FETCH JOIN
- Can specify multiple attribute paths
- Good for dynamic fetching strategies

Example:

```
@EntityGraph(attributePaths = {"books", "publisher", "reviews"})
List<Author> findAllWithDetails();
```

Solution 3: Batch Fetching

Configure Batch Size

```
@Entity
@BatchSize(size = 10)
public class Author {
    @OneToMany(mappedBy = "author")
    @BatchSize(size = 10)
    private List<Book> books;
}
```

Result: Fewer queries (1 + N/10)

Example:

- 100 authors without batch: 101 queries
- 100 authors with batch size 10: 11 queries

When to use:

- When you can't use FETCH JOIN
- Reduces queries but doesn't eliminate them
- Good middle ground

Solution 4: DTO Projection

Only Fetch What You Need

```
@Query("SELECT new AuthorDTO(a.id, a.name, COUNT(b)) " +
       "FROM Author a LEFT JOIN a.books b GROUP BY a.id, a.name")
List<AuthorDTO> findAllAuthorsWithBookCount();
```

When to use:

- When you don't need full entities
- For reporting/dashboards
- Best performance

Benefits:

- Only retrieves required fields
- No lazy loading issues
- Perfect for read-only views

Performance Impact Comparison

With 100 Authors

Approach	Database Round Trips	Performance
N+1 Problem	101 queries	❌ Slowest
Batch Fetch (size=10)	11 queries	⚠️ Better
FETCH JOIN	1 query	✅ Best
EntityGraph	1 query	✅ Best
DTO Projection	1 query	✅ Best

Performance improvement: ~100x faster with proper solution!

Best Practices

DO ✅

1. **Always use FETCH JOIN or EntityGraph** when you know you'll need associated data
2. **Keep lazy fetching as default** - only fetch what you need
3. **Monitor SQL logs** in development to catch N+1 issues early
4. **Use batch fetching** when FETCH JOIN isn't possible
5. **Consider DTOs** for read-only operations
6. **Test with realistic data volumes** - N+1 problems get worse with more data

Common Mistakes to Avoid

DON'T ❌

1. **Changing FetchType.LAZY to FetchType.EAGER globally**
    - This causes other performance issues
    - Use FETCH JOIN instead
2. **Ignoring N+1 in development**
    - "It works with 5 records" doesn't mean it works with 5000
3. **Using findAll() when you need associations**
    - Always use custom queries with FETCH JOIN
4. **Forgetting DISTINCT with FETCH JOIN**
    - Can cause duplicate results with OneToMany

Console Output Comparison

N+1 Problem Output ❌

```
Hibernate: select a1_0.id,a1_0.email,a1_0.name from authors a1_0
Hibernate: select b1_0.author_id,b1_0.id,b1_0.isbn,b1_0.title
           from books b1_0 where b1_0.author_id=?
Hibernate: select b1_0.author_id,b1_0.id,b1_0.isbn,b1_0.title
           from books b1_0 where b1_0.author_id=?
...
```

FETCH JOIN Solution Output ✅

```
Hibernate: select distinct a1_0.id,b1_0.author_id,b1_0.id,
           b1_0.isbn,b1_0.title,a1_0.email,a1_0.name
           from authors a1_0
           left join books b1_0 on a1_0.id=b1_0.author_id
```

Real-World Application

In Our Banking Application

```
// ❌ BAD: N+1 Problem
@Query("SELECT u FROM User u")
List<User> findAll();

// When accessing accounts: N queries
for (User user : users) {
    user.getAccounts().size(); // Triggers query per user!
}

// ✅ GOOD: FETCH JOIN
@Query("SELECT DISTINCT u FROM User u LEFT JOIN FETCH u.accounts")
List<User> findAllWithAccounts();

// Only 1 query for everything!
```

Bonus: Testing N+1 Detection

Hibernate Statistics

```
# Enable statistics
spring.jpa.properties.hibernate.generate_statistics=true
```

```
@Autowired
private SessionFactory sessionFactory;

@Test
public void testNoNPlusOne() {
    Statistics stats = sessionFactory.getStatistics();
    stats.clear();

    List<Author> authors = authorRepository.findAllWithBooks();

    // Assert only 1 query was executed
    assertEquals(1, stats.getPrepareStatementCount());
}
```

Key Takeaways: N+1 Query Problem

Remember

🎯 **Always monitor SQL logs** in development

🎯 **Use FETCH JOIN or EntityGraph** when loading associations

🎯 **Lazy loading is good** - but fetch eagerly when needed

🎯 **Test with realistic data** - problems scale with data size

🎯 **DTOs for reports** - best performance for read-only data

The Golden Rule

If you're loading a collection in a loop, you probably have an N+1 problem!

Bonus: JPA Query Methods Deep Dive

Complete Guide to Querying with Spring Data JPA

What is JPA?

Java Persistence API

JPA is a specification for accessing, persisting, and managing data between Java objects and relational databases.

Spring Data JPA builds on top of JPA, providing:

- Repository abstraction
- Automatic query generation
- Reduced boilerplate code
- Type-safe queries

Key Benefits:

- Write less code
- Focus on business logic
- Database independence
- Multiple query approaches

The Four Ways to Query

Spring Data JPA Query Methods

1. **Derived Query Methods**
    - Spring generates queries from method names
    - Simple and intuitive
    - No SQL/JPQL needed
2. **JPQL (Java Persistence Query Language)**
    - Object-oriented query language
    - Works with entities, not tables
    - Database-independent
3. **Native SQL Queries**
    - Direct SQL queries
    - Database-specific features
    - Maximum control
4. **Criteria API**
    - Programmatic query building
    - Type-safe
    - Dynamic queries

Method 1: Derived Query Methods

Spring Generates Queries from Method Names

```
public interface StudentRepository extends JpaRepository<Student, Long> {

    // Find by single field
    Optional<Student> findByEmail(String email);

    // Find by multiple fields (AND)
    List<Student> findByFirstNameAndLastName(String first, String last);

    // Case insensitive search
    List<Student> findByFirstNameIgnoreCase(String firstName);

    // Pattern matching
    List<Student> findByFirstNameContaining(String pattern);
    List<Student> findByFirstNameStartingWith(String prefix);

    // Comparison operators
    List<Student> findByDateOfBirthAfter(LocalDate date);
    List<Student> findByDateOfBirthBetween(LocalDate start, LocalDate end);
}
```

Derived Query Keywords

Commonly Used Keywords

Keyword	Example	SQL Equivalent
`findBy`	`findByEmail(String email)`	`WHERE email = ?`
`And`	`findByFirstNameAndLastName`	`WHERE first_name = ? AND last_name = ?`
`Or`	`findByFirstNameOrLastName`	`WHERE first_name = ? OR last_name = ?`
`Between`	`findByAgeBetween`	`WHERE age BETWEEN ? AND ?`
`LessThan`	`findByAgeLessThan`	`WHERE age < ?`
`GreaterThan`	`findByAgeGreaterThan`	`WHERE age > ?`
`Like`	`findByNameLike`	`WHERE name LIKE ?`
`Containing`	`findByNameContaining`	`WHERE name LIKE '%?%'`
`StartingWith`	`findByNameStartingWith`	`WHERE name LIKE '?%'`
`EndingWith`	`findByNameEndingWith`	`WHERE name LIKE '%?'`

Derived Query Keywords (cont.)

Keyword	Example	SQL Equivalent
`In`	`findByStatusIn(List<Status>)`	`WHERE status IN (?)`
`IsNull`	`findByPhoneNumberIsNull()`	`WHERE phone_number IS NULL`
`IsNotNull`	`findByPhoneNumberIsNotNull()`	`WHERE phone_number IS NOT NULL`
`OrderBy`	`findByStatusOrderByNameAsc`	`WHERE status = ? ORDER BY name ASC`
`Count`	`countByStatus`	`SELECT COUNT(*) WHERE status = ?`
`Exists`	`existsByEmail`	`SELECT EXISTS(WHERE email = ?)`
`Delete`	`deleteByStatus`	`DELETE WHERE status = ?`

Complete list: Spring Data JPA Reference

Method 2: JPQL Queries

Java Persistence Query Language

JPQL is an object-oriented query language that works with entities instead of tables.

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {

    // Simple JPQL query
    @Query("SELECT s FROM Student s WHERE s.firstName = :firstName")
    List<Student> findByFirstName(@Param("firstName") String firstName);

    // JPQL with multiple parameters
    @Query("SELECT s FROM Student s " +
           "WHERE s.firstName = :first AND s.lastName = :last")
    List<Student> findByFullName(@Param("first") String firstName,
                                 @Param("last") String lastName);

    // JPQL with LIKE
    @Query("SELECT s FROM Student s " +
           "WHERE LOWER(s.email) LIKE LOWER(CONCAT('%', :domain, '%'))")
    List<Student> findByEmailDomain(@Param("domain") String domain);
}
```

JPQL with JOIN FETCH

Solving the N+1 Problem

```
// ❌ BAD: N+1 Problem
@Query("SELECT s FROM Student s WHERE s.id = :id")
Optional<Student> findById(@Param("id") Long id);

// Accessing s.getEnrollments() triggers N additional queries!

// ✅ GOOD: JOIN FETCH
@Query("SELECT DISTINCT s FROM Student s " +
       "LEFT JOIN FETCH s.enrollments " +
       "WHERE s.id = :id")
Optional<Student> findByIdWithEnrollments(@Param("id") Long id);

// Everything loaded in ONE query!
```

Key Points:

- `JOIN FETCH` eagerly loads associated entities
- Use `DISTINCT` to avoid duplicates with `OneToMany`
- Solves N+1 problem completely

JPQL Aggregation Functions

COUNT, AVG, SUM, MAX, MIN

```
// Count students by gender and status
@Query("SELECT COUNT(s) FROM Student s " +
       "WHERE s.gender = :gender AND s.status = :status")
long countByGenderAndStatus(@Param("gender") Gender gender,
                            @Param("status") Status status);

// Calculate average grade for a student
@Query("SELECT AVG(e.grade) FROM Enrollment e " +
       "WHERE e.student.id = :studentId AND e.grade IS NOT NULL")
Double calculateAverageGrade(@Param("studentId") Long studentId);

// Find maximum enrollment
@Query("SELECT MAX(c.currentEnrollment) FROM Course c")
Integer findMaxEnrollment();
```

JPQL with Pagination

Using Pageable

```
@Query("SELECT s FROM Student s WHERE s.status = :status")
Page<Student> findByStatus(@Param("status") Status status,
                           Pageable pageable);

// Usage in service:
public Page<Student> getActiveStudents(int page, int size) {
    Pageable pageable = PageRequest.of(page, size,
                                       Sort.by("firstName").ascending());
    return studentRepository.findByStatus(Status.ACTIVE, pageable);
}
```

Response includes:

- Content (list of results)
- Total pages
- Total elements
- Current page number
- Has next/previous

Method 3: Native SQL Queries

Direct SQL for Database-Specific Features

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {

    // Simple native query
    @Query(value = "SELECT * FROM students WHERE date_of_birth > :date",
           nativeQuery = true)
    List<Student> findYoungerThan(@Param("date") LocalDate date);

    // Complex join query
    @Query(value = "SELECT s.* FROM students s " +
                   "JOIN enrollments e ON s.id = e.student_id " +
                   "WHERE e.course_id = :courseId",
           nativeQuery = true)
    List<Student> findStudentsByCourse(@Param("courseId") Long courseId);
}
```

Note: `nativeQuery = true` tells Spring this is SQL, not JPQL

Native SQL with Database Functions

PostgreSQL-Specific Example

```
// Use PostgreSQL's AGE function
@Query(value = "SELECT * FROM students " +
               "WHERE EXTRACT(YEAR FROM AGE(CURRENT_DATE, date_of_birth)) " +
               "BETWEEN :minAge AND :maxAge",
       nativeQuery = true)
List<Student> findByAge(@Param("minAge") int minAge,
                        @Param("maxAge") int maxAge);

// Complex aggregation with GROUP BY
@Query(value = "SELECT c.course_name, COUNT(e.id) as total, " +
               "AVG(e.grade) as avg_grade " +
               "FROM enrollments e " +
               "JOIN courses c ON e.course_id = c.id " +
               "GROUP BY c.id, c.course_name " +
               "ORDER BY avg_grade DESC",
       nativeQuery = true)
List<Object[]> getCourseStatistics();
```

Method 4: Criteria API

Type-Safe Programmatic Queries

```
@Service
public class StudentService {

    @PersistenceContext
    private EntityManager entityManager;

    public List<Student> findByStatus(Status status) {
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<Student> query = cb.createQuery(Student.class);
        Root<Student> student = query.from(Student.class);

        query.select(student)
             .where(cb.equal(student.get("status"), status));

        return entityManager.createQuery(query).getResultList();
    }
}
```

Components:

- `CriteriaBuilder` - Factory for creating query parts
- `CriteriaQuery` - The query definition
- `Root` - The FROM clause (entity)

Criteria API: Multiple Conditions

AND / OR Operations

```
public List<Student> findByGenderAndStatus(Gender gender, Status status) {
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<Student> query = cb.createQuery(Student.class);
    Root<Student> student = query.from(Student.class);

    Predicate genderPredicate = cb.equal(student.get("gender"), gender);
    Predicate statusPredicate = cb.equal(student.get("status"), status);

    query.select(student)
         .where(cb.and(genderPredicate, statusPredicate));

    return entityManager.createQuery(query).getResultList();
}
```

Predicate Operations:

- `cb.and(pred1, pred2, ...)` - AND condition
- `cb.or(pred1, pred2, ...)` - OR condition
- `cb.not(predicate)` - NOT condition

Criteria API: LIKE Query

Pattern Matching

```
public List<Student> searchByName(String pattern) {
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<Student> query = cb.createQuery(Student.class);
    Root<Student> student = query.from(Student.class);

    Predicate firstNameLike = cb.like(
        cb.lower(student.get("firstName")),
        "%" + pattern.toLowerCase() + "%"
    );

    Predicate lastNameLike = cb.like(
        cb.lower(student.get("lastName")),
        "%" + pattern.toLowerCase() + "%"
    );

    query.select(student)
         .where(cb.or(firstNameLike, lastNameLike));

    return entityManager.createQuery(query).getResultList();
}
```

Criteria API: Dynamic Queries

Building Queries with Optional Parameters

```
public List<Student> search(String firstName, String lastName,
                            Gender gender, Status status) {
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<Student> query = cb.createQuery(Student.class);
    Root<Student> student = query.from(Student.class);

    List<Predicate> predicates = new ArrayList<>();

    if (firstName != null && !firstName.isEmpty()) {
        predicates.add(cb.like(cb.lower(student.get("firstName")),
                               "%" + firstName.toLowerCase() + "%"));
    }

    if (lastName != null && !lastName.isEmpty()) {
        predicates.add(cb.like(cb.lower(student.get("lastName")),
                               "%" + lastName.toLowerCase() + "%"));
    }

    if (gender != null) {
        predicates.add(cb.equal(student.get("gender"), gender));
    }

    if (status != null) {
        predicates.add(cb.equal(student.get("status"), status));
    }

    query.select(student)
         .where(cb.and(predicates.toArray(new Predicate[0])));

    return entityManager.createQuery(query).getResultList();
}
```

Criteria API: JOIN Query

Querying Related Entities

```
public List<Student> findEnrolledInCourse(Long courseId) {
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<Student> query = cb.createQuery(Student.class);
    Root<Student> student = query.from(Student.class);

    // Join student -> enrollments -> course
    Join<Object, Object> enrollments = student.join("enrollments");
    Join<Object, Object> course = enrollments.join("course");

    query.select(student)
         .where(cb.equal(course.get("id"), courseId))
         .distinct(true);

    return entityManager.createQuery(query).getResultList();
}
```

JOIN Types:

- `join()` - INNER JOIN
- `leftJoin()` - LEFT OUTER JOIN
- `rightJoin()` - RIGHT OUTER JOIN

Criteria API: Ordering

ORDER BY Clause

```
public List<Student> findAllOrderedByName() {
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<Student> query = cb.createQuery(Student.class);
    Root<Student> student = query.from(Student.class);

    query.select(student)
         .orderBy(
             cb.asc(student.get("firstName")),
             cb.asc(student.get("lastName"))
         );

    return entityManager.createQuery(query).getResultList();
}
```

Ordering Functions:

- `cb.asc(expression)` - Ascending order
- `cb.desc(expression)` - Descending order

Criteria API: Aggregation

COUNT, AVG, SUM, MAX, MIN

```
// Count students by status
public Long countByStatus(Status status) {
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<Long> query = cb.createQuery(Long.class);
    Root<Student> student = query.from(Student.class);

    query.select(cb.count(student))
         .where(cb.equal(student.get("status"), status));

    return entityManager.createQuery(query).getSingleResult();
}
```

Aggregation Functions:

- `cb.count()` - Count rows
- `cb.avg()` - Average value
- `cb.sum()` - Sum values
- `cb.max()` - Maximum value
- `cb.min()` - Minimum value

UPDATE and DELETE Queries

Using @Modifying Annotation

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {

    // UPDATE query
    @Modifying
    @Query("UPDATE Student s SET s.status = :newStatus " +
           "WHERE s.status = :oldStatus")
    int updateStatus(@Param("oldStatus") Status oldStatus,
                     @Param("newStatus") Status newStatus);

    // DELETE query
    @Modifying
    @Query("DELETE FROM Student s " +
           "WHERE s.status = :status AND s.createdAt < :date")
    int deleteInactive(@Param("status") Status status,
                       @Param("date") LocalDateTime date);
}
```

Important: Always use with `@Transactional` in service layer!

@Modifying with @Transactional

Service Layer Implementation

```
@Service
@RequiredArgsConstructor
public class StudentService {

    private final StudentRepository studentRepository;

    @Transactional
    public int bulkUpdateStatus(Status oldStatus, Status newStatus) {
        // Returns number of rows affected
        return studentRepository.updateStatus(oldStatus, newStatus);
    }

    @Transactional
    public int cleanupInactiveStudents() {
        LocalDateTime cutoffDate = LocalDateTime.now().minusYears(2);
        return studentRepository.deleteInactive(Status.INACTIVE, cutoffDate);
    }
}
```

Why @Transactional is Required:

- Ensures data integrity
- Automatic rollback on errors
- Flushes changes to database

Comparison of Query Methods

When to Use Each Approach

Method	Best For	Pros	Cons
Derived Queries	Simple queries	Easy, no code	Limited complexity
JPQL	Complex business logic	Database-independent	Learning curve
Native SQL	DB-specific features	Full SQL power	Not portable
Criteria API	Dynamic queries	Type-safe	Verbose

Comparison: Code Examples

Same Query, Four Ways

Find students by first name:

```
// 1. Derived Query
List<Student> findByFirstName(String firstName);

// 2. JPQL
@Query("SELECT s FROM Student s WHERE s.firstName = :name")
List<Student> findByFirstNameJPQL(@Param("name") String firstName);

// 3. Native SQL
@Query(value = "SELECT * FROM students WHERE first_name = :name",
       nativeQuery = true)
List<Student> findByFirstNameNative(@Param("name") String firstName);

// 4. Criteria API
CriteriaQuery<Student> query = cb.createQuery(Student.class);
Root<Student> student = query.from(Student.class);
query.select(student).where(cb.equal(student.get("firstName"), firstName));
```

Performance Considerations

Best Practices

✅ **Use JOIN FETCH** for associations to avoid N+1 problems

✅ **Pagination** for large result sets

✅ **Projections** to fetch only needed columns

✅ **Batch fetching** with `@BatchSize`

✅ **Query logging** to monitor performance

❌ **Avoid** `FetchType.EAGER` globally

❌ **Don't** load unnecessary associations

❌ **Don't** use `findAll()` for large tables

Projections: Fetch Only What You Need

Interface-Based Projections

```
// Define projection interface
public interface StudentNameOnly {
    String getFirstName();
    String getLastName();
    String getEmail();
}

// Use in repository
@Query("SELECT s.firstName as firstName, s.lastName as lastName, " +
       "s.email as email FROM Student s WHERE s.status = :status")
List<StudentNameOnly> findNamesByStatus(@Param("status") Status status);
```

Benefits:

- Reduces data transfer
- Improves query performance
- Perfect for read-only views

DTO Projections

Class-Based Projections

```
// DTO class
public class StudentDTO {
    private Long id;
    private String fullName;
    private int enrollmentCount;

    public StudentDTO(Long id, String firstName, String lastName,
                      Long enrollmentCount) {
        this.id = id;
        this.fullName = firstName + " " + lastName;
        this.enrollmentCount = enrollmentCount.intValue();
    }
    // Getters
}

// Repository query
@Query("SELECT new com.learning.jpa.dto.StudentDTO(" +
       "s.id, s.firstName, s.lastName, COUNT(e)) " +
       "FROM Student s LEFT JOIN s.enrollments e " +
       "GROUP BY s.id, s.firstName, s.lastName")
List<StudentDTO> findStudentSummaries();
```

Hands-On Example Project

JPA Learning Application

Located in `/jpa-learning` folder of this repository.

Features:

- Complete CRUD operations
- All four query methods demonstrated
- Entity relationships (Student, Course, Enrollment)
- REST API endpoints for testing
- Comprehensive examples

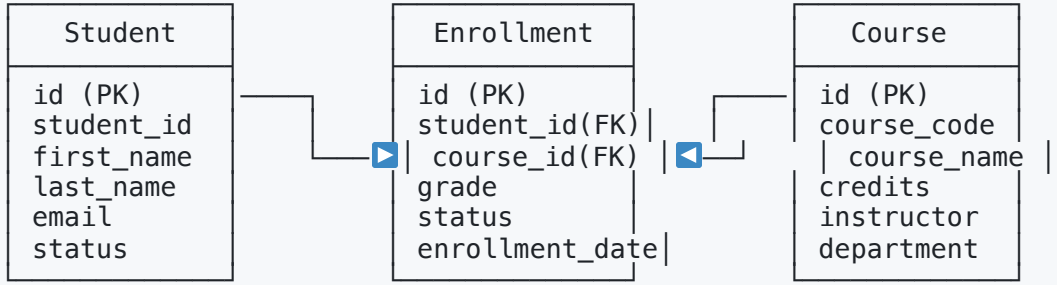
Database: `jpa_learning_db`

Port: `8001`

Endpoints: `/api/students/*`

JPA Learning: Entity Relationships

Database Schema



Relationships:

- Student (1) ———— (N) Enrollment
- Course (1) ———— (N) Enrollment

JPA Learning: Sample Endpoints

Testing Different Query Methods

```
# Derived Query Method
GET /api/students/by-email?email=john@example.com

# JPQL Query
GET /api/students/by-full-name?firstName=John&lastName=Doe

# Native SQL Query
GET /api/students/younger-than?date=2000-01-01

# Criteria API Query
GET /api/students/criteria/search?firstName=John&status=ACTIVE

# JOIN FETCH (solve N+1)
GET /api/students/123/with-enrollments

# Bulk Update
PATCH /api/students/bulk-update-status?oldStatus=ACTIVE&newStatus=GRADUATED
```

JPA Learning: Run Instructions

Quick Start

```
# 1. Create database
createdb jpa_learning_db

# 2. Navigate to project
cd jpa-learning

# 3. Run application
./mvnw spring-boot:run

# 4. Application starts on http://localhost:8001
```

Verify Setup:

```
curl http://localhost:8001/api/students
```

Key Takeaways: JPA Query Methods

Remember

🎯 **Start Simple**: Use derived queries for basic operations

🎯 **JPQL for Business Logic**: Database-independent, entity-based

🎯 **Native SQL When Needed**: Database-specific features, complex queries

🎯 **Criteria API for Dynamic**: Type-safe, programmatic queries

🎯 **Always JOIN FETCH**: Avoid N+1 problems

🎯 **Use Projections**: Fetch only what you need

🎯 **@Transactional Required**: For @Modifying queries

🎯 **Test with Real Data**: Performance issues appear with scale

Common JPA Mistakes to Avoid

DON'T ❌

1. **Using FetchType.EAGER everywhere**
    - Causes performance issues
    - Use LAZY + JOIN FETCH instead
2. **Ignoring N+1 problems**
    - Always monitor SQL logs
    - Use JOIN FETCH proactively
3. **Forgetting @Transactional on @Modifying**
    - Causes runtime errors
    - Data integrity issues
4. **Loading entire tables with findAll()**
    - Use pagination
    - Apply filters
5. **Not using projections for reports**
    - Unnecessary data loading
    - Poor performance

JPA Resources

Documentation & Learning

📚 **Official Documentation:**

- Spring Data JPA Reference
- Hibernate Documentation

📖 **Example Project:**

- `/jpa-learning` - Comprehensive examples
- README.md - Detailed guide
- All four query methods demonstrated

🔍 **Testing:**

- REST endpoints for each query type
- Sample data setup
- Performance comparisons

Questions?

Thank You!

Happy Coding! 🚀

Remember:

- Clean architecture matters
- Security is not optional
- Test your code
- Monitor in production
- Keep learning!

Contact & Resources

📧 Questions?

Check the comprehensive documentation in the repository

💻 Code Examples

All code available at: `/src/main/java/com/user/account/app/`

🧪 Testing

Use `api-tests.http` for quick testing

📖 Full Tutorial

Read `MEDIUM_ARTICLE.md` for detailed explanations

Now go build amazing Spring Boot applications! 💪