Seal Wallet - Technical Interview Guide

Mini Money Transfer Application for Brac Bank

Candidate: [Your Name]
Position: Software Engineer
Date: January 15, 2026
Interviewer: Uttam Adhikari, Head of Core Banking

Table of Contents

---

1. Project Overview & Architecture

What I Built

A **production-ready mini money transfer application** with enterprise-grade security, implementing:

- **User Management** with phone-based authentication
- **Digital Wallet** system with balance tracking
- **Money Transfer** with ACID transaction guarantees
- **JWT Authentication** with refresh token rotation
- **Comprehensive Audit Trail** for all financial operations

Technical Architecture

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   Presentation  │    │    Business     │    │   Data Access   │
│     Layer       │    │     Logic       │    │     Layer       │
```

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│ @RestController │───▶│    @Service     │───▶│   @Repository   │
│ - AuthController│    │ - UserService   │    │ - UserRepo      │
│ - WalletCtrl    │    │ - WalletService │    │ - WalletRepo    │
│ - TransactionCtrl│   │ - TransService  │    │ - TransRepo     │
└─────────────────┘    └─────────────────┘    └─────────────────┘
         │                      │                      │
         ▼                      ▼                      ▼
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   Security      │    │   Transaction   │    │   PostgreSQL    │
│   Layer         │    │   Management    │    │   Database      │
├─────────────────┤    ├─────────────────┤    ├─────────────────┤
│ JWT Filter      │    │ @Transactional  │    │ HikariCP Pool   │
│ Spring Security │    │ ACID Compliance │    │ Connection Mgmt │
│ BCrypt Hashing  │    │ Rollback Safety │    │ 5 Tables        │
└─────────────────┘    └─────────────────┘    └─────────────────┘
```

Layered Architecture Benefits

- **Separation of Concerns**: Each layer has distinct responsibilities
- **Maintainability**: Easy to modify business logic without affecting presentation
- **Testability**: Each layer can be unit tested independently
- **Scalability**: Can scale individual layers based on load

---

2. Technology Stack & Justifications

Core Technologies

Technology	Version	Why This Version?	Why Not Others?
Java	21 LTS	Latest LTS with modern features (Records, Pattern Matching, Virtual Threads)	Java 8/11 lack modern features; Java 17 is older LTS
Spring Boot	3.5.9	Latest stable with Java 21 support, enhanced security	Spring Boot 2.x doesn’t support Java 21
PostgreSQL	11.22	ACID compliance, JSON support, banking industry standard	MySQL lacks advanced features; NoSQL not suitable for financial data
Maven	3.9+	Industry standard, better dependency management	Gradle has steeper learning curve

Spring Boot Starters Used

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

    <!-- Data Access -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

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

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

    <!-- JWT Implementation -->
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-api</artifactId>
        <version>0.11.5</version>
    </dependency>
</dependencies>
```

Why These Specific Versions?

Spring Boot 3.5.9: - Native Java 21 support - Enhanced security features - Better performance with virtual threads - Jakarta EE migration complete

JJWT 0.11.5: - Latest stable JWT library - Better security than older versions - Supports modern encryption algorithms

---

3. Spring Boot Deep Dive

What is Spring Boot?

Spring Boot is an **opinionated framework** built on top of Spring Framework that: - **Eliminates boilerplate configuration** - **Provides auto-configuration** - **Includes embedded servers** - **Offers production-ready features**

Spring Boot vs Spring Framework

Aspect	Spring Framework	Spring Boot
Configuration	XML/Java Config required	Auto-configuration
Server	External server needed	Embedded Tomcat/Jetty
Dependencies	Manual dependency management	Starter dependencies
Production	Manual setup	Built-in actuator, metrics
Development	Complex setup	Rapid development

@SpringBootApplication Annotation

```
@SpringBootApplication
public class SealApplication {
    public static void main(String[] args) {
        SpringApplication.run(SealApplication.class, args);
    }
}
```

This annotation combines three annotations:

1. **@Configuration** - Marks class as configuration source
2. **@EnableAutoConfiguration** - Enables Spring Boot’s auto-configuration
3. **@ComponentScan** - Scans for components in current package and sub-packages

Key Spring Boot Features Used

1. Auto-Configuration

```
# Spring Boot automatically configures:
# - HikariCP connection pool (detected PostgreSQL driver)
# - JPA/Hibernate (detected spring-boot-starter-data-jpa)
# - Security (detected spring-boot-starter-security)
# - Jackson JSON processing (detected @RestController)
```

2. Externalized Configuration

```
# application.properties
spring.datasource.url=jdbc:postgresql://localhost:5432/seal_db
jwt.secret=sealSecretKeyForJWTTokenGenerationAndValidation2024
jwt.expiration=900000
```

3. Embedded Server

- **Tomcat 10.1.50** embedded by default
- No need for external server deployment
- Production-ready with proper configuration

---

4. Database & Connection Management

Why HikariCP?

HikariCP is Spring Boot's default connection pool because:

Feature	HikariCP	Tomcat JDBC Pool	C3P0
Performance	Fastest	Good	Slow
Memory Usage	Lowest	Medium	High
Reliability	Excellent	Good	Fair
Maintenance	Active	Active	Inactive
Spring Boot Default	✅ Yes	❌ No	❌ No

Connection Configuration

```
# HikariCP Configuration (auto-configured)
spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.connection-timeout=30000
spring.datasource.hikari.idle-timeout=600000
spring.datasource.hikari.max-lifetime=1800000
```

Database Schema Design

```sql
-- Users Table (Authentication)
CREATE TABLE users (
    id BIGSERIAL PRIMARY KEY,
    phone VARCHAR(15) UNIQUE NOT NULL,
    password TEXT NOT NULL,
    role VARCHAR(20) DEFAULT 'USER',
    status VARCHAR(20) DEFAULT 'ACTIVE',
    last_login TIMESTAMP,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Wallets Table (Financial Data)
CREATE TABLE wallets (
    id BIGSERIAL PRIMARY KEY,
    user_id BIGINT UNIQUE REFERENCES users(id) ON DELETE CASCADE,
    balance NUMERIC(15,2) DEFAULT 0.0,
    status VARCHAR(20) DEFAULT 'ACTIVE',
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Transactions Table (Audit Trail)
CREATE TABLE transactions (
    id BIGSERIAL PRIMARY KEY,
    from_wallet BIGINT REFERENCES wallets(id) ON DELETE CASCADE,
    to_wallet BIGINT REFERENCES wallets(id) ON DELETE CASCADE,
    amount NUMERIC(15,2) NOT NULL,
    type VARCHAR(20) NOT NULL,
    status VARCHAR(20) DEFAULT 'SUCCESS',
    created_at TIMESTAMP DEFAULT NOW()
);

-- Refresh Tokens Table (Security)
CREATE TABLE refresh_tokens (
    id BIGSERIAL PRIMARY KEY,
    token VARCHAR(255) UNIQUE NOT NULL,
    user_id BIGINT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    expires_at TIMESTAMP NOT NULL,
    revoked BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT NOW()
);
```

PostgreSQL Data Types Used

Type	Usage	Why?
BIGSERIAL	Primary keys	Auto-incrementing, handles large datasets
VARCHAR(n)	Phone, status	Variable length, space efficient
TEXT	Passwords, tokens	Unlimited length for hashed data
NUMERIC(15,2)	Money amounts	Exact precision for financial calculations
TIMESTAMP	Date/time fields	Timezone-aware, precise timing
BOOLEAN	Flags	True/false states

---

5. Security Implementation

Spring Security Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.csrf(csrf -> csrf.disable())
            .authorizeHttpRequests(authz -> authz
                .requestMatchers("/auth/**").permitAll()
                .anyRequest().authenticated()
            )
            .sessionManagement(session ->
                session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

        http.addFilterBefore(jwtAuthenticationFilter,
                             UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }
}
```

Authentication & Authorization Flow

```
1. User Login
   ├── Phone + Password validation
   ├── BCrypt password verification
   ├── Generate Access Token (15 min)
   ├── Generate Refresh Token (7 days)
   └── Return both tokens

2. API Request
   ├── Extract Bearer token from header
   ├── Validate JWT signature
   ├── Check token expiration
   ├── Load user details
   └── Set SecurityContext

3. Token Refresh
   ├── Validate refresh token
   ├── Check if revoked/expired
   ├── Generate new access token
   ├── Rotate refresh token
   └── Return new tokens

4. Logout
   ├── Revoke all user refresh tokens
   └── Client discards access token
```

JWT Implementation

```java
@Component
public class JwtUtil {

    @Value("${jwt.secret}")
    private String secret;

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

    public String generateToken(String phone) {
        return Jwts.builder()
                .setSubject(phone)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + expiration))
                .signWith(getSigningKey(), SignatureAlgorithm.HS256)
                .compact();
    }
}
```

Password Security

```java
@Service
public class UserService {

    @Autowired
    private PasswordEncoder passwordEncoder; // BCrypt

    public User registerUser(String phone, String password) {
        user.setPassword(passwordEncoder.encode(password)); // BCrypt hashing
        return userRepository.save(user);
    }

    public boolean validatePassword(String rawPassword, String encodedPassword) {
        return passwordEncoder.matches(rawPassword, encodedPassword);
    }
}
```

Why BCrypt?

- **Adaptive hashing** - can increase rounds as hardware improves
- **Salt included** - prevents rainbow table attacks
- **Industry standard** - used by major financial institutions
- **Spring Security default** - well-tested and maintained

---

6. Transaction Management

@Transactional Implementation

```
@Service
public class TransactionService {

    @Transactional
    public Transaction transferMoney(String fromPhone, String toPhone, BigDecimal amount)
     {
        // ACID Transaction ensures:
        // Atomicity: All operations succeed or all fail
        // Consistency: Database constraints maintained
        // Isolation: Concurrent transactions don't interfere
        // Durability: Committed changes persist

        Wallet fromWallet = walletService.findByUserPhone(fromPhone);
        Wallet toWallet = walletService.findByUserPhone(toPhone);

        // Validation
        if (!walletService.hasSufficientBalance(fromWallet, amount)) {
            throw new RuntimeException("Insufficient balance");
        }

        // Atomic operations
        fromWallet.setBalance(fromWallet.getBalance().subtract(amount));
        toWallet.setBalance(toWallet.getBalance().add(amount));

        walletRepository.save(fromWallet);
        walletRepository.save(toWallet);

        // Audit trail
        Transaction transaction = new Transaction(fromWallet, toWallet, amount, "SEND");
        return transactionRepository.save(transaction);
    }
}
```

Transaction Security Features

1. **ACID Compliance**
    - **Atomicity**: Either all operations complete or none
    - **Consistency**: Database constraints enforced
    - **Isolation**: Concurrent transactions handled safely
    - **Durability**: Committed transactions persist
2. **Business Validations**
    - Balance verification before transfer
    - Wallet status checks (ACTIVE/BLOCKED)
    - Self-transfer prevention
    - Amount validation (positive, within limits)
3. **Audit Trail**
    - Every transaction recorded with timestamp
    - Immutable transaction history
    - Complete money flow tracking

Why @Transactional is Secure

```java
// If any operation fails, entire transaction rolls back
@Transactional
public void transferMoney() {
    deductFromSender();    // Step 1
    addToReceiver();       // Step 2 - if this fails
    createAuditRecord();   // Step 3 - never executes
    // Step 1 is automatically rolled back
}
```

---

7. Logging & Exception Handling

SLF4J Logging Implementation

```java
@RestController
public class TransactionController {

    private static final Logger logger =
        LoggerFactory.getLogger(TransactionController.class);

    @PostMapping("/transfer")
    public ResponseEntity<?> transferMoney(@RequestBody TransferRequest request,
                                         Authentication authentication) {
        try {
            String fromPhone = authentication.getName();
            logger.info("Transfer request: from={}, to={}, amount={}",
                       fromPhone, request.getToPhone(), request.getAmount());

            Transaction transaction = transactionService.transferMoney(
                fromPhone, request.getToPhone(), request.getAmount());

            logger.info("Transfer successful: transactionId={}", transaction.getId());
            return ResponseEntity.ok(createResponse(transaction));

        } catch (Exception e) {
            logger.error("Transfer failed for user {}: {}",
                        authentication.getName(), e.getMessage(), e);
            return ResponseEntity.badRequest().body("Transfer failed: " + e.getMessage());
        }
    }
}
```

Why SLF4J?

Feature	SLF4J	Log4j Direct	java.util.logging
Facade Pattern	✅ Yes	❌ No	❌ No
Performance	✅ Lazy evaluation	❌ String concat	❌ String concat
Flexibility	✅ Any implementation	❌ Locked to Log4j	❌ Limited
Spring Boot Default	✅ Yes	❌ No	❌ No

Exception Handling Strategy

```java
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>> handleValidationExceptions(
            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 new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<Map<String, String>> handleRuntimeException(RuntimeException ex) {
        Map<String, String> error = new HashMap<>();
        error.put("error", ex.getMessage());
        return new ResponseEntity<>(error, HttpStatus.BAD_REQUEST);
    }
}
```

Logging Levels Used

- **ERROR**: System failures, transaction failures
- **WARN**: Security issues, invalid attempts
- **INFO**: Business operations (login, transfer, logout)
- **DEBUG**: Technical details (token validation, database queries)

Caching Strategy

Current Implementation: No caching (financial data requires real-time accuracy)

Future Considerations: - **Redis** for session management - **Application-level caching** for user profiles - **Database query caching** for read-heavy operations

---

8. Common Interview Questions & Answers

Q1: Tell us what you built. Give technical brief, describe the architecture.

Answer: I built a production-ready mini money transfer application using Spring Boot 3.5.9 with Java 21. The architecture follows a layered approach with clear separation of concerns:

- **Presentation Layer**: REST controllers handling HTTP requests/responses
- **Business Layer**: Service classes containing business logic and validations
- **Data Access Layer**: JPA repositories for database operations
- **Security Layer**: JWT-based authentication with refresh token rotation

The application implements ACID transactions for money transfers, comprehensive audit trails, and enterprise-grade security patterns used in banking systems.

Q2: Why did you choose HikariCP for database connection? Why not Tomcat JDBC Pool?

Answer: HikariCP is Spring Boot’s default connection pool and the industry standard because:

1. **Performance**: Fastest connection pool available (benchmarked)
2. **Memory Efficiency**: Lowest memory footprint
3. **Reliability**: Zero-downtime connection management
4. **Active Maintenance**: Continuously updated and optimized
5. **Spring Boot Integration**: Auto-configured with sensible defaults

Tomcat JDBC Pool is good but HikariCP outperforms it in all metrics. Since Spring Boot chose HikariCP as default after extensive testing, I leveraged that decision.

Q3: What does @Autowired do? How did you handle transactions? Are transactions secure?

Answer:

@Autowired: Enables dependency injection by automatically wiring beans from Spring container. Spring creates and manages object lifecycle, promoting loose coupling.

Transaction Handling: I used `@Transactional` annotation for declarative transaction management:

```java
@Transactional
public Transaction transferMoney(String fromPhone, String toPhone, BigDecimal amount) {
    // All operations are atomic - either all succeed or all rollback
}
```

Transaction Security: Yes, transactions are secure because: - **ACID compliance** ensures data consistency - **Automatic rollback** on any failure prevents partial updates - **Isolation levels** prevent concurrent transaction interference - **Business validations** prevent invalid operations - **Audit trails** track all financial operations

Q4: How did you implement Spring Security?

Answer: I implemented a comprehensive security system:

1. **JWT Authentication**: Stateless token-based authentication
2. **Refresh Token Rotation**: Enterprise-grade token management
3. **BCrypt Password Hashing**: Industry-standard password security
4. **Method-level Security**: Protected endpoints with role-based access
5. **CSRF Protection**: Disabled for stateless API (appropriate for REST APIs)

The security flow: Login → JWT + Refresh Token → API calls with JWT → Token refresh when expired → Logout revokes refresh tokens.

Q5: How does authentication/authorization work in your project?

Answer:

Authentication Flow: 1. User submits phone + password 2. System validates credentials against BCrypt hash 3. Generates JWT access token (15 min) + refresh token (7 days) 4. Client includes JWT in Authorization header for API calls 5. JWT filter validates token on each request

Authorization: Currently role-based (USER role), but architecture supports fine-grained permissions. All financial endpoints require authentication.

Token Security: Refresh token rotation prevents token replay attacks, and server-side revocation enables immediate logout.

Q6: What are you using for logging, caching, exception handling?

Answer:

Logging: SLF4J with Logback (Spring Boot default) - Structured logging with parameterized messages - Different levels: ERROR, WARN, INFO, DEBUG - Security-aware (no sensitive data in logs)

Caching: Currently none (financial data requires real-time accuracy) - Future: Redis for session management, application-level caching for user profiles

Exception Handling: Global exception handler with `@RestControllerAdvice` - Validation errors mapped to field-specific messages - Runtime exceptions handled gracefully - Consistent error response format

Q7: Which server are you using?

Answer: Embedded Tomcat 10.1.50 (Spring Boot default). Benefits:

- **No external server needed**: Self-contained JAR deployment
- **Production-ready**: Handles thousands of concurrent connections
- **Auto-configured**: Optimal settings out of the box
- **Easy deployment**: Single JAR file deployment
- **Monitoring**: Built-in metrics and health checks

For production, can easily switch to Jetty or Undertow if needed.

Q8: How is your Java knowledge?

Answer: I’m proficient in modern Java (using Java 21 LTS):

Core Strengths: - **OOP Principles**: Inheritance, Polymorphism, Encapsulation, Abstraction - **Collections Framework**: Lists, Maps, Sets with appropriate implementations - **Exception Handling**: Try-catch, custom exceptions, best practices - **Multithreading**: Concurrent programming, thread safety - **Modern Features**: Streams, Lambda expressions, Optional, Records

Java 21 Features Used: - **Pattern Matching**: Enhanced switch expressions - **Records**: Immutable data classes for DTOs - **Text Blocks**: Readable SQL queries - **Virtual Threads**: Future scalability (not yet implemented)

Q9: What does @SpringBootApplication annotation do?

Answer: `@SpringBootApplication` is a composite annotation combining:

1. **@Configuration**: Marks class as configuration source for beans
2. **@EnableAutoConfiguration**: Triggers Spring Boot’s auto-configuration magic
3. **@ComponentScan**: Scans current package and sub-packages for components

It essentially bootstraps the entire Spring Boot application with minimal configuration.

Q10: What is Spring Boot vs Spring Framework?

Answer:

Aspect	Spring Framework	Spring Boot
Purpose	Comprehensive framework	Rapid application development
Configuration	Manual XML/Java config	Auto-configuration
Dependencies	Manual management	Starter dependencies
Server	External deployment	Embedded server
Production	Manual setup	Built-in actuator

Spring Boot is built on Spring Framework but eliminates boilerplate configuration and provides opinionated defaults for faster development.

Q11: Key Spring Boot features you used?

Answer:

1. **Auto-Configuration**: Automatic HikariCP, JPA, Security setup
2. **Starter Dependencies**: Web, Data JPA, Security, Validation starters
3. **Embedded Server**: Tomcat for easy deployment
4. **Externalized Configuration**: Properties file for environment-specific settings
5. **Actuator**: Health checks and metrics (can be added)
6. **DevTools**: Hot reload during development
7. **Profile Support**: Different configurations for dev/prod

Q12: What are Spring Boot Starters?

Answer: Starters are dependency descriptors that bring in all necessary dependencies for a specific functionality:

- **spring-boot-starter-web**: Web applications (Tomcat, Spring MVC, Jackson)
- **spring-boot-starter-data-jpa**: JPA/Hibernate (Hibernate, HikariCP, Transaction management)
- **spring-boot-starter-security**: Security (Spring Security, BCrypt)
- **spring-boot-starter-validation**: Bean validation (Hibernate Validator)

Benefits: Consistent versions, reduced dependency conflicts, faster setup.

Q13: @Controller vs @RestController - which did you use and why?

Answer: I used `@RestController` because:

@RestController = @Controller + @ResponseBody

- **@Controller**: Returns view names (for web pages)
- **@RestController**: Returns data directly (for APIs)

Since I’m building a REST API (not web pages), `@RestController` automatically serializes return objects to JSON, which is exactly what I need for API responses.

Q14: Explain Auto-Configuration in Spring Boot.

Answer: Auto-configuration automatically configures beans based on:

1. **Classpath Dependencies**: If H2 is on classpath → configure H2 datasource
2. **Existing Beans**: If DataSource bean exists → skip datasource auto-config
3. **Property Values**: If `spring.datasource.url` set → use that URL
4. **Conditional Annotations**: `@ConditionalOnClass`, `@ConditionalOnProperty`

Example: Spring Boot sees PostgreSQL driver → auto-configures HikariCP → sets up JPA → configures transaction manager.

Q15: How does your project support externalized configuration?

Answer: Through `application.properties`:

```
# Database Configuration
spring.datasource.url=jdbc:postgresql://localhost:5432/seal_db
spring.datasource.username=seal_admin
spring.datasource.password=seal

# JWT Configuration
```

```
jwt.secret=sealSecretKeyForJWTTokenGenerationAndValidation2024
jwt.expiration=900000

# Server Configuration
server.port=8080
```

Benefits: - **Environment-specific**: Different configs for dev/test/prod - **Security**: Sensitive data externalized - **Flexibility**: Change behavior without code changes - **Profile Support**: `application-{profile}.properties`

Q16: What is Spring Boot Actuator and why is it useful?

Answer: Actuator provides production-ready features:

Endpoints: - `/actuator/health`: Application health status - `/actuator/metrics`: Performance metrics - `/actuator/info`: Application information - `/actuator/env`: Environment properties

Benefits for Banking: - **Monitoring**: Real-time application health - **Metrics**: Transaction throughput, response times - **Debugging**: Thread dumps, heap dumps - **Security**: Audit trails, security events

Production Usage: Essential for monitoring financial applications where uptime and performance are critical.

Additional Banking-Specific Questions

Q17: How would you handle high transaction volumes?

Answer:

1. **Database Optimization**:
   - Connection pooling (HikariCP already implemented)
   - Database indexing on frequently queried columns
   - Read replicas for reporting queries
2. **Application Scaling**:
   - Horizontal scaling with load balancers
   - Stateless design (JWT) enables easy scaling
   - Microservices architecture for different domains
3. **Caching Strategy**:
   - Redis for session management
   - Application-level caching for user profiles
   - Database query result caching
4. **Async Processing**:
   - Message queues for non-critical operations
   - Event-driven architecture
   - Batch processing for reports

Q18: How do you ensure data consistency in distributed systems?

Answer:

1. **ACID Transactions**: Already implemented for single-database operations
2. **Distributed Transactions**: Two-phase commit or Saga pattern
3. **Event Sourcing**: Immutable event log for audit trails
4. **Eventual Consistency**: For non-critical operations
5. **Database Constraints**: Foreign keys, check constraints
6. **Application-level Validation**: Business rule enforcement

Q19: What security measures would you add for production?

Answer:

1. **Enhanced Authentication**:
   - Multi-factor authentication (MFA)
   - Biometric authentication
   - Device fingerprinting
2. **API Security**:
   - Rate limiting to prevent abuse
   - API gateway for centralized security
   - Request/response encryption
3. **Monitoring & Auditing**:
   - Real-time fraud detection
   - Comprehensive audit logs
   - Security event monitoring
4. **Infrastructure Security**:
   - HTTPS/TLS encryption
   - Network segmentation
   - Regular security updates

Q20: How would you implement regulatory compliance (PCI DSS, etc.)?

Answer:

1. **Data Protection**:
   - Encryption at rest and in transit
   - PII data masking in logs
   - Secure key management
2. **Access Control**:
   - Role-based access control (RBAC)
   - Principle of least privilege
   - Regular access reviews
3. **Audit Requirements**:
   - Immutable audit trails
   - Log retention policies
   - Compliance reporting
4. **Testing & Validation**:
   - Regular penetration testing
   - Code security scanning
   - Compliance audits

Conclusion

This Seal Wallet application demonstrates enterprise-grade development practices suitable for banking systems:

- **Robust Architecture**: Layered design with clear separation of concerns
- **Security First**: JWT with refresh token rotation, BCrypt hashing
- **Data Integrity**: ACID transactions, comprehensive validations
- **Production Ready**: Proper logging, exception handling, monitoring
- **Modern Technology**: Java 21, Spring Boot 3.5.9, PostgreSQL

The implementation follows banking industry best practices and can serve as a foundation for larger financial systems.

Prepared by: [Your Name]
Date: January 15, 2026
Contact: [Your Contact Information]