SESAME UNIVERSITY

Secure Programming Exam Statement

AI-Powered Secure Financial Intelligence System

for Tunisian Economic Challenges

Using Spring Boot, Spring Security 6.5, Spring AI, and Java 25

Prepared by:

Chaouki Bayoudhi

Academic Year 2025-2026

December 14, 2025

Project Title

Secure AI-Powered Financial Fraud Detection and Risk Management System for Tunisian SMEs with Intelligent Health Monitoring

Project Context and Motivation

Tunisia's economy faces significant challenges in the financial sector, particularly for Small and Medium Enterprises (SMEs). Financial fraud, transaction security, credit risk assessment, and economic volatility pose serious threats to business sustainability. Traditional financial systems lack the sophistication to detect complex fraud patterns, assess real-time risks, and provide intelligent insights for decision-making.

Modern secure programming practices combined with cutting-edge AI technologies offer unprecedented capabilities for building robust, secure financial systems. This project requires students to develop a comprehensive Spring Boot application that integrates advanced security mechanisms, AI-powered fraud detection, risk analysis using multiple AI libraries, and an intelligent health monitoring system supervised by an AI agent.

Why is this critical for Tunisia?

- SMEs represent over 90% of Tunisian businesses and are vulnerable to financial fraud
- Digital payment fraud has increased significantly, requiring advanced detection systems
- Credit risk assessment is crucial for financial institutions and business sustainability
- Real-time fraud detection can prevent significant financial losses
- AI-powered insights help businesses make data-driven financial decisions
- Secure coding practices protect sensitive financial data from cyber threats
- Health monitoring ensures system reliability and early anomaly detection

Project Objectives

The main objectives of this project are:

- Design and implement a secure Spring Boot application following security best practices
- Apply Spring Security 6.5 for authentication, authorization, and protection against common vulnerabilities
- Integrate Spring AI for LLM-powered financial analysis and report generation
- Implement fraud detection using multiple AI libraries: Deep Java Library (DJL), ONNX Runtime, TensorFlow Java
- Build an intelligent health monitoring system supervised by an AI agent

- Demonstrate modern Java 25 features: records, enumerations, annotations, interfaces, functional interfaces
- Apply proper inheritance, polymorphism, encapsulation, and abstraction principles
- Use Lombok for code simplification and maintainability
- Implement comprehensive exception handling and error management
- Create secure REST APIs with proper input validation and output sanitization
- Apply OWASP Top 10 security practices throughout the application
- Design and implement secure software architecture patterns

Problem Statement

Build a secure, AI-powered financial intelligence system that:

1. Detects fraudulent financial transactions in real-time using multiple AI models
2. Assesses credit risk for Tunisian SMEs based on financial data and market conditions
3. Generates intelligent financial reports using LLM integration (LangChain4j, GPT-J, or local models)
4. Monitors system health through an AI agent that analyzes metrics and generates alerts
5. Protects sensitive financial data using Spring Security 6.5 and secure coding practices
6. Provides secure REST APIs for financial data management and analysis

What does this mean in practical terms?

- You will build a Spring Boot application with multiple microservices/modules
- Each transaction will be analyzed by AI models to detect fraud patterns
- The system will assess creditworthiness using historical data and AI predictions
- An AI agent will continuously monitor system health and performance metrics
- All endpoints will be secured using Spring Security 6.5 with JWT tokens
- Financial data will be encrypted at rest and in transit
- The system will generate intelligent reports using LLM capabilities
- You must follow secure coding practices to prevent common vulnerabilities

System Architecture and Security Requirements

High-Level Architecture

The system must be designed with security-first principles:

- **Presentation Layer:** Secure REST APIs with input validation and output encoding
- **Business Logic Layer:** Service classes with proper authorization checks
- **Data Access Layer:** Secure database access with parameterized queries
- **AI/ML Layer:** Multiple AI models for fraud detection and risk assessment
- **LLM Integration Layer:** Spring AI with LangChain4j for intelligent analysis
- **Health Monitoring Layer:** AI agent monitoring system metrics and generating alerts
- **Security Layer:** Spring Security 6.5 with JWT, OAuth2, and role-based access control

Security Requirements (Critical)

1. **Authentication:** JWT-based authentication with refresh tokens, secure password hashing (BCrypt/Argon2)
2. **Authorization:** Role-based access control (RBAC) with fine-grained permissions
3. **Input Validation:** Comprehensive validation using Bean Validation (Jakarta Validation) and custom validators
4. **SQL Injection Prevention:** Use Spring Data JPA with parameterized queries, never use raw SQL
5. **XSS Prevention:** Output encoding and Content Security Policy (CSP) headers
6. **CSRF Protection:** CSRF tokens for state-changing operations
7. **Data Encryption:** Encrypt sensitive data at rest (AES-256) and in transit (TLS 1.3)
8. **Secrets Management:** Use environment variables or secure vaults, never hardcode secrets
9. **Error Handling:** Secure error messages that don't leak sensitive information
10. **Logging and Auditing:** Comprehensive audit logs for security events
11. **Rate Limiting:** Implement rate limiting to prevent brute force attacks
12. **API Security:** API key management and request signing for external integrations

Core Features and Implementation Requirements

Feature 1: Secure User Management

- Implement user registration with email verification
- Secure password policies (minimum length, complexity requirements)
- Password reset functionality with secure tokens
- User roles: ADMIN, FINANCIAL_ANALYST, SME_USER, AUDITOR
- Account lockout after failed login attempts
- Session management with configurable timeout

Required Java Components:

- `User` entity class with Lombok annotations
- `UserRole` enumeration (ADMIN, FINANCIAL_ANALYST, SME_USER, AUDITOR)
- `UserRepository` interface extending JpaRepository
- `UserService` interface and implementation
- `UserController` with proper security annotations
- Custom validation annotations for password strength
- `UserRegistrationRequest` and `UserResponse` records

Feature 2: Financial Transaction Management

- Secure transaction creation, retrieval, and update
- Transaction validation and business rule enforcement
- Transaction history with pagination and filtering
- Support for multiple transaction types: PAYMENT, TRANSFER, WITHDRAWAL, DEPOSIT
- Transaction status tracking: PENDING, COMPLETED, FAILED, FRAUD_DETECTED

Required Java Components:

- `Transaction` entity with proper relationships
- `TransactionType` enumeration
- `TransactionStatus` enumeration
- `TransactionRepository` with custom query methods

- `TransactionService` interface and implementation
- `TransactionController` with role-based access
- `TransactionRequest` and `TransactionResponse` records
- Custom exceptions: `InsufficientFundsException`, `InvalidTransactionException`

Feature 3: AI-Powered Fraud Detection

- Real-time fraud detection using multiple AI models
- Integration with Deep Java Library (DJL) for neural network models
- ONNX Runtime integration for pre-trained fraud detection models
- TensorFlow Java for custom fraud detection models
- Ensemble approach combining predictions from multiple models
- Fraud score calculation and threshold-based classification
- Fraud pattern learning from historical data

Required Java Components:

- `FraudDetectionService` interface
- `DJLFraudDetector` implementation using Deep Java Library
- `ONNXFraudDetector` implementation using ONNX Runtime
- `TensorFlowFraudDetector` implementation using TensorFlow Java
- `FraudDetectionResult` record with confidence scores
- `FraudPattern` entity for storing detected patterns
- `FraudDetectionStrategy` functional interface
- Custom exceptions: `ModelLoadException`, `InferenceException`

Feature 4: Credit Risk Assessment

- AI-powered credit risk scoring for SMEs
- Integration with multiple ML models for risk prediction
- Historical financial data analysis
- Market condition integration
- Risk category classification: LOW, MEDIUM, HIGH, CRITICAL

- Risk report generation with recommendations

Required Java Components:

- `CreditRiskService` interface
- `RiskAssessment` record with risk score and category
- `RiskCategory` enumeration
- `FinancialData` record for input data
- `RiskModel` interface for different risk models
- `HistoricalRiskAnalyzer` implementation
- `RiskReport` record with detailed analysis

Feature 5: LLM-Powered Financial Intelligence

- Integration with Spring AI for LLM capabilities
- LangChain4j integration for local or remote LLM models
- GPT-J integration for financial report generation
- Automated financial analysis report generation
- Market trend analysis using LLM
- Intelligent recommendations for financial decisions
- Support for local models (Ollama, JLama) via LangChain4j

Required Java Components:

- `FinancialIntelligenceService` interface
- `LLMService` interface for LLM interactions
- `LangChain4jService` implementation
- `GPTJService` implementation for GPT-J integration
- `ReportGenerator` service using Spring AI
- `FinancialReport` record
- `LLMConfig` configuration class
- Custom exceptions: `LLMServiceException`, `ReportGenerationException`

Feature 6: AI Agent Health Monitoring System

- Continuous system health monitoring
- AI agent that analyzes system metrics
- Automatic anomaly detection in system performance
- Health status classification: HEALTHY, WARNING, CRITICAL, DOWN
- Intelligent alert generation based on patterns
- Predictive health analysis using ML models
- Automated remediation suggestions

Required Java Components:

- `HealthMonitoringService` interface
- `HealthAgent` class implementing AI agent logic
- `SystemMetrics` record for collecting metrics
- `HealthStatus` enumeration
- `HealthAlert` record with severity and recommendations
- `MetricsCollector` interface
- `AnomalyDetector` using AI models
- `HealthReport` record with comprehensive analysis
- Scheduled tasks using `@Scheduled` annotation

Feature 7: Secure Dashboard with Statistics and Advanced Functionalities

- Comprehensive dashboard displaying real-time financial statistics and KPIs
- Interactive data visualization with charts and graphs
- Role-based dashboard views (different data for different user roles)
- Real-time data updates using WebSocket or Server-Sent Events (SSE)
- Advanced filtering, sorting, and search capabilities
- Export functionality for reports (PDF, Excel, CSV)
- Responsive design supporting multiple devices
- Secure data aggregation and statistical calculations

Required Complex Functionalities (Choose 3-4):

1. **Advanced Fraud Analytics Dashboard:**
   - Real-time fraud detection metrics and trends
   - Fraud pattern visualization with interactive charts
   - Geographic fraud heatmap showing fraud distribution
   - Time-series analysis of fraud incidents
   - Fraud prediction forecasting using AI models
   - Comparative analysis (fraud rates by transaction type, user category, etc.)
   - Drill-down capabilities for detailed fraud investigation
2. **Financial Performance Analytics:**
   - Revenue and transaction volume analytics
   - Profit/loss trends and forecasting
   - Cash flow analysis and visualization
   - Financial health score calculation and display
   - Comparative performance metrics (month-over-month, year-over-year)
   - Budget vs. actual analysis
   - Financial ratio calculations (liquidity, profitability, etc.)
   - Interactive financial reports with drill-down capabilities
3. **Risk Assessment Dashboard:**
   - Real-time risk score monitoring and visualization
   - Risk distribution across different SME categories
   - Risk trend analysis with predictive indicators
   - Portfolio risk analysis and diversification metrics
   - Risk alert system with severity-based notifications
   - Historical risk pattern analysis
   - Risk mitigation recommendations display
   - Interactive risk heatmap by geographic region or business sector
4. **User Activity and Security Monitoring Dashboard:**
   - User activity tracking and analytics
   - Login patterns and suspicious activity detection
   - Failed authentication attempts monitoring
   - Session management and active user tracking
   - Security event log visualization
   - Access pattern analysis (time-based, location-based)

- User behavior anomaly detection
- Security compliance metrics and reporting

5. **AI Model Performance Dashboard:**
    - Model accuracy metrics and performance indicators
    - Prediction confidence distribution visualization
    - Model comparison across different AI frameworks (DJL, ONNX, TensorFlow)
    - Training data quality metrics
    - Model drift detection and alerts
    - Inference latency and throughput monitoring
    - A/B testing results for different model versions
    - Model explainability metrics and feature importance visualization

Required Java Components:

- `DashboardService` interface for data aggregation
- `StatisticsService` interface for statistical calculations
- `DashboardController` with secure endpoints
- `DashboardDTO` records for dashboard data transfer
- `ChartData` record for visualization data
- `StatisticsAggregator` utility class
- `DashboardSecurityFilter` for role-based data filtering
- `RealTimeDataService` for WebSocket/SSE implementation
- `ExportService` interface for report generation
- `DashboardConfig` configuration class
- Custom exceptions: `DashboardDataException`, `StatisticsCalculationException`

Security Requirements for Dashboard:

- Role-based data access (users only see data they're authorized to view)
- Data masking for sensitive financial information
- Rate limiting on dashboard endpoints to prevent abuse
- Secure WebSocket connections (WSS) for real-time updates
- Input validation for all dashboard filters and parameters
- Output encoding to prevent XSS attacks
- Audit logging for dashboard access and data exports
- CORS configuration for dashboard API endpoints

Step-by-Step Implementation Guide

Step 1: Project Setup and Dependencies

1. Create a new Spring Boot 3.3+ project using Spring Initializr or your IDE
2. Configure Java 25 (or latest LTS version) as the project SDK
3. Add the following dependencies to `pom.xml` or `build.gradle`:
    - Spring Boot Starter Web
    - Spring Boot Starter Security (6.5+)
    - Spring Boot Starter Data JPA
    - Spring Boot Starter Validation
    - Spring AI (latest version)
    - PostgreSQL Driver (or H2 for development)
    - Lombok
    - JWT libraries (io.jsonwebtoken:jjwt-api, jjwt-impl, jjwt-jackson)
    - Deep Java Library (DJL) dependencies
    - ONNX Runtime Java API
    - TensorFlow Java
    - LangChain4j
    - Spring Boot Actuator (for health monitoring)
4. Configure application properties for database, security, and AI services
5. Set up proper package structure:

```
com.tunisia.financial
  +-- config
  +-- controller
  +-- service
  +-- repository
  +-- entity
  +-- dto
  +-- security
  +-- ai
  +-- exception
  +-- util
  +-- monitoring
```

Step 2: Security Configuration

1. Create `SecurityConfig` class with `@Configuration` and `@EnableWebSecurity`
2. Configure JWT authentication filter
3. Set up password encoder (BCryptPasswordEncoder or Argon2PasswordEncoder)
4. Configure CORS policies
5. Set up CSRF protection
6. Configure method security with `@EnableMethodSecurity`
7. Create `JwtTokenProvider` utility class
8. Implement `JwtAuthenticationFilter` extending `OncePerRequestFilter`
9. Create `CustomUserDetailsService` implementing `UserDetailsService`
10. Configure security rules for public and protected endpoints

Example Security Configuration:

```java
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(
            HttpSecurity http,
            JwtAuthenticationFilter jwtAuthFilter) throws Exception {
        http
            .csrf(csrf -> csrf.disable())
            .sessionManagement(session ->
                session.sessionCreationPolicy(
                    SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/api/public/**").permitAll()
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated())
            .addFilterBefore(jwtAuthFilter,
                UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(12);
    }
}
```

Step 3: Entity and Repository Layer

1. Create base entity classes using proper inheritance
2. Implement `User` entity with Lombok annotations
3. Create `Transaction` entity with relationships
4. Implement `FraudPattern` entity
5. Create `HealthMetric` entity for monitoring
6. Use JPA annotations properly: `@Entity`, `@Table`, `@Id`, `@GeneratedValue`
7. Implement proper relationships: `@OneToMany`, `@ManyToOne`, `@ManyToMany`
8. Create repository interfaces extending `JpaRepository`
9. Add custom query methods using `@Query` annotation
10. Implement soft delete pattern using `@SQLDelete` and `@Where`

Example Entity with Records:

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

    @Enumerated(EnumType.STRING)
    private TransactionType type;

    @Enumerated(EnumType.STRING)
    private TransactionStatus status;

    @Column(nullable = false)
    @DecimalMin(value = "0.01")
    private BigDecimal amount;

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

    @Column(name = "fraud_score")
    private Double fraudScore;

    @CreatedDate
    private LocalDateTime createdAt;

    @LastModifiedDate
    private LocalDateTime updatedAt;
}
```

Step 4: DTOs and Records

1. Create request DTOs using Java records
2. Create response DTOs using Java records
3. Implement proper validation annotations
4. Use custom validators for complex validation rules
5. Create mapper interfaces or use MapStruct for DTO conversion
6. Implement proper null safety

Example Records:

```java
public record TransactionRequest(
    @NotNull(message = "Amount is required")
    @DecimalMin(value = "0.01", message = "Amount must be positive")
    BigDecimal amount,

    @NotNull(message = "Type is required")
    TransactionType type,

    @NotBlank(message = "Description is required")
    @Size(max = 500, message = "Description too long")
    String description,

    @NotNull(message = "Recipient ID is required")
    Long recipientId
) {}

public record TransactionResponse(
    Long id,
    TransactionType type,
    TransactionStatus status,
    BigDecimal amount,
    Double fraudScore,
    LocalDateTime createdAt,
    String description
) {}
```

Step 5: Service Layer with Interfaces

1. Create service interfaces following Interface Segregation Principle

2. Implement services with proper transaction management
3. Use `@Transactional` annotation appropriately
4. Implement proper exception handling
5. Use dependency injection with constructor injection
6. Apply business logic validation
7. Implement caching where appropriate using Spring Cache
8. Use functional interfaces for strategy patterns

Example Service Interface and Implementation:

```java
public interface FraudDetectionService {
    FraudDetectionResult detectFraud(Transaction transaction);
    List<FraudPattern> getFraudPatterns();
    void updateModel(String modelType);
}

@Service
@RequiredArgsConstructor
@Slf4j
public class FraudDetectionServiceImpl implements FraudDetectionService {

    private final DJLFraudDetector djlDetector;
    private final ONNXFraudDetector onnxDetector;
    private final TensorFlowFraudDetector tfDetector;

    @Override
    @Transactional(readOnly = true)
    public FraudDetectionResult detectFraud(Transaction transaction) {
        try {
            // Ensemble approach: combine multiple models
            var djlResult = djlDetector.detect(transaction);
            var onnxResult = onnxDetector.detect(transaction);
            var tfResult = tfDetector.detect(transaction);

            // Weighted average of confidence scores
            double avgConfidence = (djlResult.confidence() +
                                    onnxResult.confidence() +
                                    tfResult.confidence()) / 3.0;

            boolean isFraud = avgConfidence > 0.7;

            return new FraudDetectionResult(
                isFraud,
                avgConfidence,
                List.of(djlResult, onnxResult, tfResult)
            );
        } catch (Exception e) {
            log.error("Error detecting fraud for transaction {}",
                        transaction.getId(), e);
            throw new FraudDetectionException(
                "Failed to detect fraud", e);
        }
    }
}
```

Step 6: AI Integration - Deep Java Library (DJL)

1. Add DJL dependencies for your target engine (PyTorch, TensorFlow, or ONNX)
2. Create `DJLFraudDetector` class
3. Load pre-trained model or create model architecture
4. Implement feature extraction from transaction data
5. Perform inference using DJL's `Predictor` interface
6. Handle model loading exceptions properly
7. Implement model versioning

Example DJL Integration:

```
@Service
@Slf4j
public class DJLFraudDetector {

    private Predictor<float[], float[]> predictor;

    @PostConstruct
    public void loadModel() {
        try {
            Criteria<float[], float[]> criteria = Criteria.builder()
                .setTypes(float[].class, float[].class)
                .optModelPath(Paths.get("models/fraud_detection.onnx"))
                .optEngine("OnnxRuntime")
                .build();

            Model model = ModelZoo.loadModel(criteria);
            predictor = model.newPredictor();
            log.info("DJL fraud detection model loaded successfully");
        } catch (Exception e) {
            log.error("Failed to load DJL model", e);
            throw new ModelLoadException("DJL model loading failed", e);
        }
    }

    public FraudDetectionResult detect(Transaction transaction) {
        float[] features = extractFeatures(transaction);
        float[] prediction = predictor.predict(features);
        double confidence = prediction[0];
```

```java
        boolean isFraud = confidence > 0.5;

        return new FraudDetectionResult(isFraud, confidence, null);
    }

    private float[] extractFeatures(Transaction transaction) {
        // Extract numerical features from transaction
        return new float[]{
            transaction.getAmount().floatValue(),
            // ... other features
        };
    }
}
```

Step 7: AI Integration - ONNX Runtime

1. Add ONNX Runtime Java dependencies
2. Create `ONNXFraudDetector` class
3. Load ONNX model file
4. Prepare input tensors from transaction data
5. Run inference using ONNX Runtime
6. Extract and interpret results
7. Handle ONNX-specific exceptions

Example ONNX Integration:

```java
@Service
@Slf4j
public class ONNXFraudDetector {

    private OrtEnvironment env;
    private OrtSession session;

    @PostConstruct
    public void initialize() {
        try {
            env = OrtEnvironment.getEnvironment();
            session = env.createSession("models/fraud_model.onnx",
                new OrtSession.SessionOptions());
            log.info("ONNX model loaded successfully");
        } catch (Exception e) {
            log.error("Failed to load ONNX model", e);
            throw new ModelLoadException("ONNX model loading failed", e);
        }
    }

    public FraudDetectionResult detect(Transaction transaction) {
        try {
            OnnxTensor inputTensor = createInputTensor(transaction);
            OrtSession.Result result = session.run(
                Collections.singletonMap("input", inputTensor));

            float[][] output = (float[][]) result.get(0).getValue();
            double confidence = output[0][0];
            boolean isFraud = confidence > 0.5;

            inputTensor.close();
            result.close();

            return new FraudDetectionResult(isFraud, confidence, null);
        } catch (Exception e) {
            log.error("ONNX inference failed", e);
            throw new InferenceException("ONNX inference error", e);
        }
    }
}
```

Step 8: AI Integration - TensorFlow Java

1. Add TensorFlow Java dependencies
2. Create `TensorFlowFraudDetector` class
3. Load SavedModel or frozen graph
4. Create input tensors
5. Run inference using TensorFlow Java API
6. Extract predictions
7. Handle TensorFlow-specific errors

Step 9: Spring AI and LangChain4j Integration

1. Configure Spring AI in application properties
2. Create `LLMService` interface
3. Implement LangChain4j integration for local models (Ollama, JLama)
4. Implement GPT-J integration
5. Create prompt templates for financial analysis
6. Implement report generation service
7. Handle LLM API errors and rate limiting
8. Implement response streaming if supported

Example Spring AI Integration:

```java
@Service
@RequiredArgsConstructor
@Slf4j
public class FinancialIntelligenceService {

    private final ChatClient chatClient;

    public FinancialReport generateReport(
            List<Transaction> transactions,
            RiskAssessment riskAssessment) {

        String prompt = buildAnalysisPrompt(transactions, riskAssessment);

        try {
            String analysis = chatClient.call(prompt);

            return FinancialReport.builder()
                .summary(extractSummary(analysis))
                .recommendations(extractRecommendations(analysis))
                .riskFactors(extractRiskFactors(analysis))
                .generatedAt(LocalDateTime.now())
                .build();
        } catch (Exception e) {
            log.error("Failed to generate financial report", e);
            throw new ReportGenerationException(
                "LLM report generation failed", e);
        }
    }

    private String buildAnalysisPrompt(
            List<Transaction> transactions,
            RiskAssessment riskAssessment) {
        return String.format("""
            Analyze the following financial data for a Tunisian SME:

            Transaction Count: %d
            Total Volume: %s
            Risk Category: %s
            Risk Score: %.2f

            Provide:
            1. Financial health summary
            2. Risk factors identified
            3. Recommendations for improvement
            4. Market outlook for Tunisian SMEs
            """,
            transactions.size(),
            calculateTotalVolume(transactions),
            riskAssessment.category(),
            riskAssessment.score());
    }
}
```

Example LangChain4j Integration:

```java
@Service
@Slf4j
public class LangChain4jService implements LLMService {

    private final ChatLanguageModel chatModel;

    public LangChain4jService() {
        // Configure for local Ollama
        this.chatModel = OllamaChatModel.builder()
            .baseUrl("http://localhost:11434")
            .modelName("llama2")
            .temperature(0.7)
            .build();
    }

    @Override
    public String generateAnalysis(String prompt) {
        try {
            Response<AiMessage> response = chatModel.generate(prompt);
            return response.content().text();
        } catch (Exception e) {
            log.error("LangChain4j generation failed", e);
            throw new LLMServiceException("Analysis generation failed", e);
        }
    }
}
```

Step 10: Health Monitoring with AI Agent

1. Create `HealthMonitoringService` interface
2. Implement `HealthAgent` class with AI capabilities
3. Collect system metrics using Spring Actuator
4. Create `SystemMetrics` record
5. Implement anomaly detection using ML models
6. Generate health alerts with severity levels
7. Implement scheduled health checks using `@Scheduled`
8. Create health dashboard endpoint (secured)
9. Integrate LLM for intelligent health analysis

Example Health Monitoring:

```java
@Service
@RequiredArgsConstructor
@Slf4j
public class HealthMonitoringService {

    private final MeterRegistry meterRegistry;
    private final HealthAgent healthAgent;
    private final LLMService llmService;

    @Scheduled(fixedRate = 60000) // Every minute
    public void monitorHealth() {
        SystemMetrics metrics = collectMetrics();
        HealthStatus status = healthAgent.analyze(metrics);

        if (status != HealthStatus.HEALTHY) {
            HealthAlert alert = generateAlert(metrics, status);
            sendAlert(alert);

            // Use LLM for intelligent analysis
            String analysis = llmService.analyzeHealthIssue(metrics, status);
            log.warn("Health issue detected: {}", analysis);
        }
    }

    private SystemMetrics collectMetrics() {
        return SystemMetrics.builder()
            .cpuUsage(getCpuUsage())
            .memoryUsage(getMemoryUsage())
            .diskUsage(getDiskUsage())
            .activeConnections(getActiveConnections())
            .requestRate(getRequestRate())
            .errorRate(getErrorRate())
            .timestamp(LocalDateTime.now())
            .build();
    }
}

@Component
@Slf4j
public class HealthAgent {

    private final AnomalyDetector anomalyDetector;

    public HealthStatus analyze(SystemMetrics metrics) {
        // Use ML model to detect anomalies
        boolean hasAnomaly = anomalyDetector.detect(metrics);

        if (hasAnomaly) {
            double severity = calculateSeverity(metrics);
            if (severity > 0.9) return HealthStatus.CRITICAL;
            if (severity > 0.7) return HealthStatus.WARNING;
        }

        return HealthStatus.HEALTHY;
    }

    private double calculateSeverity(SystemMetrics metrics) {
        // Calculate severity based on multiple factors
        double cpuSeverity = metrics.cpuUsage() > 90 ? 1.0 :
                           metrics.cpuUsage() / 90.0;
        double memorySeverity = metrics.memoryUsage() > 85 ? 1.0 :
                               metrics.memoryUsage() / 85.0;

        return Math.max(cpuSeverity, memorySeverity);
    }
}
```

Step 11: Dashboard Implementation with Statistics

1. Create `DashboardService` interface and implementation
2. Implement `StatisticsService` for data aggregation and calculations
3. Create dashboard DTOs (records) for data transfer:
   - `DashboardSummary` record
   - `ChartData` record
   - `StatisticsResponse` record
4. Implement 3-4 complex dashboard functionalities from the list:
   - Advanced Fraud Analytics Dashboard
   - Financial Performance Analytics
   - Risk Assessment Dashboard
   - User Activity and Security Monitoring Dashboard
   - AI Model Performance Dashboard
5. Create `DashboardController` with secure endpoints
6. Implement role-based data filtering in `DashboardSecurityFilter`
7. Add real-time data updates using WebSocket or Server-Sent Events
8. Implement export functionality (PDF, Excel, CSV) using `ExportService`
9. Create statistical calculation utilities:
   - Time-series aggregation
   - Comparative analysis calculations
   - Trend analysis and forecasting

- Percentage and ratio calculations

10. Implement caching for dashboard data to improve performance
11. Add input validation for all dashboard filters and parameters
12. Create dashboard configuration class

Example Dashboard Service:

```java
@Service
@RequiredArgsConstructor
@Slf4j
public class DashboardServiceImpl implements DashboardService {

    private final TransactionRepository transactionRepository;
    private final FraudDetectionService fraudService;
    private final CreditRiskService riskService;
    private final StatisticsService statisticsService;

    @Override
    @PreAuthorize("hasAnyRole('ADMIN', 'FINANCIAL_ANALYST')")
    public DashboardSummary getDashboardSummary(
            LocalDate startDate,
            LocalDate endDate,
            Authentication authentication) {

        // Role-based data filtering
        String username = authentication.getName();
        UserRole role = getCurrentUserRole(authentication);

        // Aggregate statistics
        TransactionStatistics txStats =
            statisticsService.calculateTransactionStats(startDate, endDate,
                role);

        FraudStatistics fraudStats =
            statisticsService.calculateFraudStats(startDate, endDate, role);

        RiskStatistics riskStats =
            statisticsService.calculateRiskStats(startDate, endDate, role);

        // Calculate KPIs
        double totalRevenue = txStats.totalAmount();
        double fraudRate = (fraudStats.fraudCount() /
                           (double) txStats.totalCount()) * 100;
        double avgRiskScore = riskStats.averageRiskScore();

        return DashboardSummary.builder()
            .totalTransactions(txStats.totalCount())
            .totalRevenue(totalRevenue)
            .fraudCount(fraudStats.fraudCount())
            .fraudRate(fraudRate)
```

```java
                .averageRiskScore(avgRiskScore)
                .highRiskCount(riskStats.highRiskCount())
                .period(startDate + " to " + endDate)
                .lastUpdated(LocalDateTime.now())
                .build();
    }

    @Override
    public List<ChartData> getFraudTrendData(
            LocalDate startDate,
            LocalDate endDate,
            ChartPeriod period) {

        // Aggregate fraud data by period
        Map<LocalDate, Long> fraudByPeriod =
            transactionRepository
                .findFraudTransactionsByPeriod(startDate, endDate)
                .stream()
                .collect(Collectors.groupingBy(
                    tx -> period.toPeriodStart(tx.getCreatedAt()),
                    Collectors.counting()
                ));

        return fraudByPeriod.entrySet().stream()
            .map(entry -> new ChartData(
                entry.getKey().toString(),
                entry.getValue().doubleValue()
            ))
            .sorted(Comparator.comparing(ChartData::label))
            .collect(Collectors.toList());
    }

    @Override
    public RiskHeatmapData getRiskHeatmapData() {
        // Geographic or sector-based risk distribution
        Map<String, Double> riskByRegion =
            riskService.getRiskDistributionByRegion();

        return RiskHeatmapData.builder()
            .dataPoints(riskByRegion.entrySet().stream()
                .map(entry -> new HeatmapPoint(
                    entry.getKey(),
                    entry.getValue()
                ))
                .collect(Collectors.toList()))
            .maxRisk(riskByRegion.values().stream()
                .max(Double::compareTo)
                .orElse(0.0))
            .minRisk(riskByRegion.values().stream()
                .min(Double::compareTo)
                .orElse(0.0))
            .build();
    }
}
```

Example Dashboard Controller:

```
@RestController
@RequestMapping("/api/dashboard")
@RequiredArgsConstructor
@Validated
@Slf4j
public class DashboardController {

    private final DashboardService dashboardService;
    private final ExportService exportService;

    @GetMapping("/summary")
    @PreAuthorize("hasAnyRole('ADMIN', 'FINANCIAL_ANALYST', 'SME_USER')")
    public ResponseEntity<DashboardSummary> getSummary(
            @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
                LocalDate startDate,
            @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
                LocalDate endDate,
            Authentication authentication) {

        validateDateRange(startDate, endDate);

        DashboardSummary summary = dashboardService.getDashboardSummary(
            startDate, endDate, authentication);

        return ResponseEntity.ok(summary);
    }

    @GetMapping("/fraud-analytics")
    @PreAuthorize("hasAnyRole('ADMIN', 'FINANCIAL_ANALYST')")
    public ResponseEntity<FraudAnalyticsResponse> getFraudAnalytics(
            @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
                LocalDate startDate,
            @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
                LocalDate endDate) {

        FraudAnalyticsResponse analytics =
            dashboardService.getFraudAnalytics(startDate, endDate);

        return ResponseEntity.ok(analytics);
    }

    @GetMapping("/export")
    @PreAuthorize("hasAnyRole('ADMIN', 'FINANCIAL_ANALYST')")
    public ResponseEntity<Resource> exportDashboard(
            @RequestParam ExportFormat format,
            @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
                LocalDate startDate,
            @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
```

```java
                LocalDate endDate,
            Authentication authentication) {

        byte[] exportData = exportService.exportDashboard(
            format, startDate, endDate, authentication);

        return ResponseEntity.ok()
            .header(HttpHeaders.CONTENT_DISPOSITION,
                   "attachment; filename=\"dashboard." + format.getExtension()
                       + "\"")
            .contentType(MediaType.parseMediaType(format.getContentType()))
            .body(new ByteArrayResource(exportData));
    }

    private void validateDateRange(LocalDate start, LocalDate end) {
        if (start.isAfter(end)) {
            throw new ValidationException("Start date must be before end
                date");
        }
        if (ChronoUnit.DAYS.between(start, end) > 365) {
            throw new ValidationException("Date range cannot exceed 365 days");
        }
    }
}
```

Step 12: Controller Layer with Security

1. Create REST controllers with proper annotations
2. Apply security annotations: `@PreAuthorize`, `@Secured`
3. Implement proper input validation using `@Valid`
4. Handle exceptions using `@ExceptionHandler`
5. Return proper HTTP status codes
6. Implement pagination and sorting
7. Add API documentation using Swagger/OpenAPI
8. Implement rate limiting

Example Secure Controller:

```java
@RestController
@RequestMapping("/api/transactions")
@RequiredArgsConstructor
@Validated
@Slf4j
public class TransactionController {

    private final TransactionService transactionService;

    @PostMapping
    @PreAuthorize("hasRole('SME_USER') or hasRole('FINANCIAL_ANALYST')")
    public ResponseEntity<TransactionResponse> createTransaction(
            @Valid @RequestBody TransactionRequest request,
            Authentication authentication) {

        String username = authentication.getName();
        TransactionResponse response =
            transactionService.createTransaction(request, username);

        return ResponseEntity.status(HttpStatus.CREATED).body(response);
    }

    @GetMapping("/{id}")
    @PreAuthorize("hasRole('SME_USER') or hasRole('FINANCIAL_ANALYST')")
    public ResponseEntity<TransactionResponse> getTransaction(
            @PathVariable Long id,
            Authentication authentication) {

        TransactionResponse response =
            transactionService.getTransaction(id, authentication.getName());

        return ResponseEntity.ok(response);
    }

    @GetMapping
    @PreAuthorize("hasRole('FINANCIAL_ANALYST') or hasRole('ADMIN')")
    public ResponseEntity<Page<TransactionResponse>> getAllTransactions(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "20") int size,
            @RequestParam(required = false) TransactionType type) {

        Page<TransactionResponse> transactions =
            transactionService.getAllTransactions(page, size, type);

        return ResponseEntity.ok(transactions);
    }

    @ExceptionHandler(ValidationException.class)
    public ResponseEntity<ErrorResponse> handleValidation(
            ValidationException ex) {
        return ResponseEntity.badRequest()
            .body(new ErrorResponse("Validation failed", ex.getMessage()));
    }
}
```

Step 13: Exception Handling

1. Create custom exception classes

2. Implement global exception handler using `@ControllerAdvice`

3. Create proper error response DTOs
4. Map exceptions to appropriate HTTP status codes
5. Log exceptions securely (don't leak sensitive info)
6. Implement exception hierarchy using inheritance

Example Exception Handling:

```java
@ControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

    @ExceptionHandler(EntityNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleNotFound(
            EntityNotFoundException ex) {
        log.warn("Entity not found: {}", ex.getMessage());
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .body(new ErrorResponse("Resource not found", ex.getMessage()));
    }

    @ExceptionHandler(SecurityException.class)
    public ResponseEntity<ErrorResponse> handleSecurity(
            SecurityException ex) {
        log.warn("Security violation: {}", ex.getMessage());
        return ResponseEntity.status(HttpStatus.FORBIDDEN)
            .body(new ErrorResponse("Access denied",
                  "You don't have permission to perform this action"));
    }

    @ExceptionHandler(ValidationException.class)
    public ResponseEntity<ErrorResponse> handleValidation(
            ValidationException ex) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST)
            .body(new ErrorResponse("Validation failed", ex.getMessage()));
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGeneric(Exception ex) {
        log.error("Unexpected error", ex);
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body(new ErrorResponse("Internal server error",
                  "An unexpected error occurred"));
    }
}

// Custom Exceptions
public class FraudDetectionException extends RuntimeException {
    public FraudDetectionException(String message) {
        super(message);
    }

    public FraudDetectionException(String message, Throwable cause) {
        super(message, cause);
    }
}

public class InsufficientFundsException extends BusinessException {
    public InsufficientFundsException(BigDecimal available, BigDecimal
        required) {
        super(String.format("Insufficient funds: available=%.2f,
            required=%.2f",
            available, required));
    }
}
```

Step 14: Testing and Validation

1. Write unit tests for services using JUnit 5 and Mockito
2. Write integration tests for controllers
3. Test security configurations
4. Test AI model integrations with mock data
5. Test exception handling
6. Perform security testing (OWASP ZAP, etc.)
7. Load testing for performance
8. Test health monitoring system

Required Java Features and Patterns

Records

Use Java records for immutable data transfer objects:

```java
public record FraudDetectionResult(
    boolean isFraud,
    double confidence,
    List<ModelResult> modelResults
) {}

public record SystemMetrics(
    double cpuUsage,
    double memoryUsage,
    double diskUsage,
    int activeConnections,
    double requestRate,
    double errorRate,
    LocalDateTime timestamp
) {}
```

Enumerations

Create type-safe enumerations:

```java
public enum TransactionType {
    PAYMENT("Payment transaction"),
    TRANSFER("Money transfer"),
    WITHDRAWAL("Cash withdrawal"),
    DEPOSIT("Cash deposit");

    private final String description;

    TransactionType(String description) {
        this.description = description;
    }

    public String getDescription() {
        return description;
    }
}

public enum HealthStatus {
    HEALTHY, WARNING, CRITICAL, DOWN
}

public enum RiskCategory {
    LOW(0.0, 0.3),
    MEDIUM(0.3, 0.6),
    HIGH(0.6, 0.8),
    CRITICAL(0.8, 1.0);

    private final double minScore;
    private final double maxScore;

    RiskCategory(double minScore, double maxScore) {
        this.minScore = minScore;
        this.maxScore = maxScore;
    }

    public static RiskCategory fromScore(double score) {
        for (RiskCategory category : values()) {
            if (score >= category.minScore && score < category.maxScore) {
                return category;
            }
        }
        return CRITICAL;
    }
}
```

Annotations

Create custom annotations for validation and security:

```java
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = StrongPasswordValidator.class)
public @interface StrongPassword {
    String message() default "Password must be strong";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("hasRole('ADMIN')")
public @interface AdminOnly {
}
```

Interfaces and Functional Interfaces

```java
// Service Interface
public interface FraudDetectionService {
    FraudDetectionResult detectFraud(Transaction transaction);
    void retrainModel();
}

// Functional Interface for Strategy Pattern
@FunctionalInterface
public interface FraudDetectionStrategy {
    FraudDetectionResult detect(Transaction transaction);
}

// Functional Interface for Metrics Collection
@FunctionalInterface
public interface MetricsCollector {
    SystemMetrics collect();
}

// Interface for AI Models
public interface AIModel {
    PredictionResult predict(double[] features);
    void loadModel(String modelPath) throws ModelLoadException;
    boolean isLoaded();
}
```

Inheritance

Demonstrate proper inheritance:

```java
// Base Exception
public abstract class BusinessException extends RuntimeException {
    protected BusinessException(String message) {
        super(message);
    }

    protected BusinessException(String message, Throwable cause) {
        super(message, cause);
    }
}

// Specific Exceptions
public class FraudDetectionException extends BusinessException {
    public FraudDetectionException(String message) {
        super(message);
    }
}

public class ModelLoadException extends BusinessException {
    public ModelLoadException(String message, Throwable cause) {
        super(message, cause);
    }
}

// Base Entity
@MappedSuperclass
@Getter
@Setter
public abstract class BaseEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @CreatedDate
    private LocalDateTime createdAt;

    @LastModifiedDate
    private LocalDateTime updatedAt;

    @Version
    private Long version;
}

// Concrete Entities
@Entity
public class Transaction extends BaseEntity {
    // Transaction-specific fields
}

@Entity
public class User extends BaseEntity {
    // User-specific fields
}
```

Lombok Usage

Use Lombok to reduce boilerplate:

```java
@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@ToString(exclude = "password")
@EqualsAndHashCode(callSuper = true)
public class User extends BaseEntity {
    @Column(unique = true, nullable = false)
    private String username;

    @Column(nullable = false)
    private String email;

    @Column(nullable = false)
    private String password;

    @Enumerated(EnumType.STRING)
    private UserRole role;
}

@Service
@RequiredArgsConstructor
@Slf4j
public class TransactionService {
    private final TransactionRepository repository;
    private final FraudDetectionService fraudService;

    // Constructor injection via @RequiredArgsConstructor
    // Logger via @Slf4j
}
```

Security Best Practices Checklist

Authentication and Authorization

Implement JWT with proper expiration and refresh tokens

Use strong password hashing (BCrypt with cost factor 12+ or Argon2)

Implement account lockout after failed attempts

Use role-based access control (RBAC)

Implement method-level security with `@PreAuthorize`

Secure password reset with time-limited tokens

Implement session timeout

Input Validation and Sanitization

Validate all user inputs using Bean Validation

Sanitize output to prevent XSS

Use parameterized queries (Spring Data JPA)

Validate file uploads (type, size, content)

Implement rate limiting on APIs

Validate and sanitize LLM inputs/outputs

Data Protection

Encrypt sensitive data at rest

Use TLS 1.3 for data in transit

Never log sensitive information (passwords, tokens, financial data)

Implement proper data masking in logs

Use secure storage for secrets (environment variables, vaults)

Implement data retention and deletion policies

Error Handling

Don't expose stack traces to users

Use generic error messages for security errors

Log errors securely with appropriate detail

Implement proper exception hierarchy

Handle AI/ML model errors gracefully

API Security

Implement CORS policies

Use HTTPS only

Implement API versioning

Add request/response logging (sanitized)

Implement API key management for external services

Add security headers (CSP, X-Frame-Options, etc.)

Technologies and Tools

Core Technologies

- **Java:** Java 25 (or latest LTS version)
- **Spring Boot:** 3.3+ (latest stable version)
- **Spring Security:** 6.5+ (latest version)
- **Spring AI:** Latest version for LLM integration
- **Spring Data JPA:** For database operations
- **PostgreSQL:** Production database (H2 for development)
- **Lombok:** For reducing boilerplate code

AI and ML Libraries

- **Deep Java Library (DJL):** For deep learning models
- **ONNX Runtime Java API:** For ONNX model inference
- **TensorFlow Java:** For TensorFlow model integration
- **LangChain4j:** For LLM integration and local models (Ollama, JLama)
- **Spring AI:** For Spring-native AI integration
- **GPT-J/NeoX:** For local LLM models (optional)

Security Libraries

- **Spring Security:** Authentication and authorization
- **JWT (io.jsonwebtoken):** For token-based authentication
- **BCrypt/Argon2:** For password hashing
- **OWASP Dependency Check:** For vulnerability scanning

Development Tools

- **Maven or Gradle:** Build tool
- **JUnit 5:** Unit testing
- **Mockito:** Mocking framework
- **Spring Boot Actuator:** Health monitoring
- **Swagger/OpenAPI:** API documentation

Project Structure

Your project should follow this package structure:

```
com.tunisia.financial
+-- config/
|   +-- SecurityConfig.java
|   +-- JwtConfig.java
|   +-- CorsConfig.java
|   +-- AiConfig.java
|   +-- CacheConfig.java
+-- controller/
|   +-- AuthController.java
|   +-- TransactionController.java
|   +-- FraudController.java
|   +-- RiskController.java
|   +-- HealthController.java
|   +-- ReportController.java
|   +-- DashboardController.java
+-- service/
|   +-- UserService.java
|   +-- UserServiceImpl.java
|   +-- TransactionService.java
|   +-- TransactionServiceImpl.java
|   +-- FraudDetectionService.java
|   +-- FraudDetectionServiceImpl.java
|   +-- CreditRiskService.java
|   +-- CreditRiskServiceImpl.java
|   +-- FinancialIntelligenceService.java
|   +-- FinancialIntelligenceServiceImpl.java
|   +-- HealthMonitoringService.java
|   +-- HealthMonitoringServiceImpl.java
|   +-- DashboardService.java
|   +-- DashboardServiceImpl.java
|   +-- StatisticsService.java
|   +-- StatisticsServiceImpl.java
|   +-- ExportService.java
|   +-- ExportServiceImpl.java
+-- repository/
|   +-- UserRepository.java
|   +-- TransactionRepository.java
|   +-- FraudPatternRepository.java
|   +-- HealthMetricRepository.java
```

```
+-- entity/
|   +-- BaseEntity.java
|   +-- User.java
|   +-- Transaction.java
|   +-- FraudPattern.java
|   +-- HealthMetric.java
+-- dto/
|   +-- request/
|   |   +-- TransactionRequest.java
|   |   +-- UserRegistrationRequest.java
|   |   +-- RiskAssessmentRequest.java
|   +-- response/
|   |   +-- TransactionResponse.java
|   |   +-- FraudDetectionResult.java
|   |   +-- RiskAssessment.java
|   |   +-- FinancialReport.java
|   |   +-- HealthStatus.java
|   |   +-- DashboardSummary.java
|   |   +-- ChartData.java
|   |   +-- FraudAnalyticsResponse.java
|   |   +-- RiskHeatmapData.java
|   +-- ErrorResponse.java
+-- security/
|   +-- JwtTokenProvider.java
|   +-- JwtAuthenticationFilter.java
|   +-- CustomUserDetailsService.java
|   +-- SecurityUtils.java
|   +-- DashboardSecurityFilter.java
+-- ai/
|   +-- fraud/
|   |   +-- DJLFraudDetector.java
|   |   +-- ONNXFraudDetector.java
|   |   +-- TensorFlowFraudDetector.java
|   |   +-- FraudDetectionStrategy.java
|   +-- risk/
|   |   +-- RiskModel.java
|   |   +-- HistoricalRiskAnalyzer.java
|   |   +-- CreditRiskCalculator.java
|   +-- llm/
|   |   +-- LLMService.java
|   |   +-- LangChain4jService.java
|   |   +-- GPTJService.java
|   |   +-- SpringAIService.java
|   +-- monitoring/
|       +-- HealthAgent.java
|       +-- AnomalyDetector.java
|       +-- MetricsCollector.java
+-- exception/
|   +-- BusinessException.java
|   +-- FraudDetectionException.java
|   +-- ModelLoadException.java
|   +-- InferenceException.java
|   +-- InsufficientFundsException.java
|   +-- ValidationException.java
|   +-- GlobalExceptionHandler.java
+-- util/
|   +-- EncryptionUtil.java
|   +-- ValidationUtil.java
|   +-- ModelConverter.java
+-- monitoring/
|   +-- HealthMonitoringService.java
|   +-- SystemMetrics.java
|   +-- HealthAlert.java
+-- FinancialApplication.java
```

Evaluation Criteria

Your project will be evaluated based on:

Security Implementation (25%)

- Proper implementation of Spring Security 6.5
- JWT authentication and authorization
- Input validation and output sanitization
- SQL injection prevention
- XSS and CSRF protection
- Secure error handling
- Secrets management
- Security best practices throughout

Code Quality and Architecture (20%)

- Proper use of Java 25 features (records, enums, annotations, interfaces)
- Correct application of OOP principles (inheritance, polymorphism, encapsulation)
- Use of Lombok appropriately
- Clean code principles
- Proper package structure
- Design patterns implementation
- Exception handling

AI Integration (15%)

- Successful integration of DJL, ONNX Runtime, TensorFlow Java
- Spring AI and LangChain4j integration
- Functional fraud detection system
- Working LLM-powered report generation
- Health monitoring with AI agent
- Proper error handling for AI services

Functionality (10%)

- All core features implemented
- REST APIs working correctly
- Database operations functioning
- Health monitoring operational
- Dashboard with statistics and complex functionalities implemented
- 3-4 complex dashboard functionalities working
- Real-time data updates functioning
- Export functionality operational
- End-to-end functionality

Dashboard Implementation (10%)

- Comprehensive dashboard with statistics and KPIs
- Implementation of 3-4 complex functionalities from the provided list
- Role-based dashboard views and data filtering
- Real-time data updates (WebSocket or SSE)
- Interactive data visualization with charts and graphs
- Export functionality (PDF, Excel, CSV)
- Secure dashboard endpoints with proper authorization
- Performance optimization (caching, efficient queries)
- User-friendly interface and responsive design

Documentation and Testing (10%)

- Code documentation (JavaDoc)
- README with setup instructions
- API documentation (Swagger/OpenAPI)
- Unit tests
- Security testing documentation

Submission Requirements

Project Type

This is an **individual project**. Each student must complete the work independently. Collaboration or sharing of code between students is strictly prohibited.

Required Deliverables

1. **Source Code:** Complete Spring Boot project with all features implemented
    - All source code files properly organized in the project structure
    - README.md with setup instructions (including dependencies, configuration, and how to run the project)
    - Proper code documentation (JavaDoc comments)
    - Unit tests and integration tests
2. **Datasets:** All datasets used in the project

- Include all datasets used for training AI models (fraud detection, risk assessment, etc.)
- Include any sample data or test datasets
- Provide documentation explaining the datasets (source, format, usage)
- If datasets are too large, include them in the Google Drive folder (see instructions below)

3. **Demo Video:** 10-15 minute video demonstrating:
   - System setup and configuration
   - Security features (authentication, authorization, JWT tokens)
   - Fraud detection in action with AI models
   - LLM-powered report generation
   - Dashboard with statistics and complex functionalities
   - Health monitoring system with AI agent
   - API testing and endpoints demonstration
   - Real-time features (if implemented)

Note: PowerPoint presentation is **NOT required**. Only source code, datasets, and demo video must be submitted.

Submission Format

- All files in a single compressed folder (ZIP or RAR format)
- Folder name: `SecureProgramming_[YourName]_[StudentID]`
- Example: `SecureProgramming_JohnDoe_20240001`
- The compressed folder should contain:
  - Complete source code directory (Spring Boot project)
  - README.md file
  - Datasets folder (or reference to datasets location)
  - Demo video file (MP4, MOV, or other common video formats)

Submission Platform

- Submit your project on **Moodle**
- Upload the compressed folder (ZIP or RAR) containing source code, datasets, and demo video
- **File Size Limitation:** If your project size exceeds Moodle's upload limits, follow these steps:
  1. Upload your complete project (source code, datasets, demo video) to **Google Drive**
  2. Share the Google Drive folder with the following email address:

     `chaouki.bayoudhi@sesame.com.tn`

3. Grant **View** or **Editor** permissions to the above email address
4. Create a text file named `GOOGLE_DRIVE_LINK.txt` containing only the Google Drive shareable link
5. Upload this `GOOGLE_DRIVE_LINK.txt` file to Moodle in the source code deposit space
6. Ensure the link is accessible and permissions are correctly set

- Verify your submission was successful before the deadline
- **Important:** Whether uploading directly to Moodle or via Google Drive, you must still create a submission entry on Moodle

Important Deadlines

Submission Deadline: January 15, 2026 (15/01/2026)

Important:

- Late submissions will **NOT** be accepted
- Submit your work well before the deadline to avoid technical issues
- Ensure all files are included and the project can be compiled and run
- Test your demo video to ensure it plays correctly

Evaluation

- Evaluation will take place during the **exam period**
- The exact evaluation date will be fixed and planned by the administration
- Students will be notified of the evaluation schedule through official channels
- During evaluation, students may be asked to:
  - Demonstrate their project functionality
  - Explain their code and implementation choices
  - Answer questions about security features and AI integration
  - Show the dashboard and complex functionalities
- Be prepared to run your project and demonstrate all features

Useful Resources

Spring Security Documentation

- Spring Security 6.5: `https://docs.spring.io/spring-security/reference/`
- OWASP Top 10: `https://owasp.org/www-project-top-ten/`
- Spring Security Best Practices: `https://spring.io/guides/topicals/spring-security-architecture`

Spring AI Documentation

- Spring AI: `https://docs.spring.io/spring-ai/reference/`
- LangChain4j: `https://github.com/langchain4j/langchain4j`

AI/ML Libraries

- Deep Java Library (DJL): `https://djl.ai/`
- ONNX Runtime Java: `https://onnxruntime.ai/docs/tutorials/java/`
- TensorFlow Java: `https://www.tensorflow.org/jvm`

Java 25 Features

- Java Records: `https://docs.oracle.com/en/java/javase/17/language/records.html`
- Java Sealed Classes: `https://openjdk.org/jeps/409`
- Pattern Matching: `https://openjdk.org/jeps/441`

Additional Notes

- **Security First:** Security is the primary focus of this exam. Ensure all security requirements are met.
- **AI Integration:** You must integrate at least 2 different AI libraries (DJL, ONNX, or TensorFlow) for fraud detection.
- **Health Monitoring:** The AI agent for health monitoring is mandatory and must demonstrate intelligent analysis.
- **Dashboard:** The dashboard with statistics and 3-4 complex functionalities is mandatory. You must choose 3-4 functionalities from the provided list and implement them with proper security, role-based access, and real-time updates.
- **Code Quality:** Write clean, maintainable, well-documented code following Java best practices.
- **Testing:** Include comprehensive unit tests and security tests.
- **Documentation:** Thorough documentation is essential for evaluation.
- **Originality:** While you may use AI tools for assistance, you must understand and be able to explain all code.
- **Scope Management:** Focus on core security features first, then extend with additional features if time permits.

Good Luck!

Be confident, be secure, be innovative!