Maven Project Complete Analysis

## Learning Management System (LMS)

# 1. Maven Dependencies (pom.xml)

Dependencies are external libraries that your project needs to function. They are automatically downloaded by Maven from the central repository.

**📦 Jackson Databind v2.15.2**

**GroupId:** com.fasterxml.jackson.core

**ArtifactId:** jackson-databind

**Purpose:** JSON Processing Library - Used for converting Java objects to JSON (serialization) and JSON back to Java objects (deserialization). This is essential for data persistence in this project.

**Used In:** All Repository classes (StudentRepository, TeacherRepository, AdminRepository, CourseRepository, etc.) use ObjectMapper from Jackson to read/write JSON files.

**Example Usage:**

// Reading from JSON file  
ObjectMapper objectMapper = new ObjectMapper();  
List<Student> students = objectMapper.readValue(  
 new File("students.json"),   
 new TypeReference<List<Student>>() {}  
);  
  
// Writing to JSON file  
objectMapper.writerWithDefaultPrettyPrinter()  
 .writeValue(new File("students.json"), students);

# 2. Maven Plugins

Plugins extend Maven's functionality. They perform specific tasks during the build lifecycle (compile, test, package, etc.).

**🔌 Plugin 1: Maven Compiler Plugin v3.11.0**

**GroupId:** org.apache.maven.plugins

**ArtifactId:** maven-compiler-plugin

**Purpose:** Compiles Java source code (.java files) into bytecode (.class files). Configured to use Java 17 as both source and target version.

**Configuration:**

<source>17</source>  
<target>17</target>  
  
Meaning:  
- source: Java version of your source code (Java 17)  
- target: Java version for compiled bytecode (Java 17)

**What it does:** Runs during the "compile" phase. Compiles all .java files in src/main/java/ to .class files in target/classes/

**🔌 Plugin 2: Exec Maven Plugin v3.1.0**

**GroupId:** org.codehaus.mojo

**ArtifactId:** exec-maven-plugin

**Purpose:** Executes Java programs directly from Maven. Allows you to run the application without creating a JAR file.

**Configuration:**

<mainClass>sms.app.Main</mainClass>  
  
This tells Maven which class contains the main() method.

**Command:**

mvn exec:java -Dexec.mainClass="sms.app.Main"  
  
Or simply: mvn exec:java (uses configured mainClass)

# 3. Maven Folder Structure

Maven follows a standard directory layout called "Convention over Configuration". This structure is recognized by all Maven projects.

project-root/  
├── pom.xml # Project configuration file  
├── src/ # Source code directory  
│ └── main/ # Main application code  
│ └── java/ # Java source files  
│ └── sms/ # Base package  
│ ├── app/ # Application layer  
│ ├── domain/ # Domain entities  
│ ├── data/ # Data access layer  
│ ├── exceptions/# Custom exceptions  
│ ├── services/ # Business logic  
│ ├── search/ # Search interfaces  
│ ├── sort/ # Sort interfaces  
│ └── validation/# Input validation  
├── target/ # Build output (generated)  
│ ├── classes/ # Compiled .class files  
│ ├── generated-sources/ # Auto-generated code  
│ └── maven-status/ # Build metadata  
├── uploads/ # File upload storage  
└── \*.json # Data persistence files

**📁 Folder Explanations:**

* **pom.xml**: Project Object Model - Maven configuration file containing dependencies, plugins, and project metadata
* **src/main/java/**: Source code directory for main application (production code)
* **src/test/java/**: Source code directory for test code (not present in this project)
* **target/**: Build output directory (auto-generated, excluded from Git). Contains compiled classes and build artifacts
* **target/classes/**: Compiled .class files (bytecode) - this is what Java actually runs
* **uploads/**: Custom directory for file upload storage (created by application)
* **\*.json files**: JSON files for data persistence (students.json, teachers.json, courses.json, etc.)

# 4. Understanding mvn exec:java Command

**What Happens When You Run:**

mvn exec:java -Dexec.mainClass="sms.app.Main"

* **Step 1**: Maven checks if source code is compiled. If not, it runs the compiler plugin.
* **Step 2**: Maven looks for the main class: sms.app.Main
* **Step 3**: Maven sets up the classpath (includes target/classes/ and all dependencies)
* **Step 4**: Maven executes: java -cp [classpath] sms.app.Main
* **Step 5**: The main() method in Main.java starts execution
* **Step 6**: Application runs until completion or termination

**Alternative Commands:**

# Run with configured mainClass (from pom.xml)  
mvn exec:java  
  
# Compile then run  
mvn clean compile exec:java  
  
# Full build and run  
mvn clean package exec:java

# 5. Main Method and Application Startup

**📄 Main Class Location:** src/main/java/sms/app/Main.java

**Package:** sms.app

**Full Class Name:** sms.app.Main

**🔄 Execution Flow:**

public static void main(String[] args) {  
 try {  
 // 1. Initialize repositories and services  
 initializeSystem();  
   
 // 2. Demonstrate OOP features  
 demonstrateOopFeatures();  
   
 // 3. Start interactive menu  
 runMainMenu();  
   
 } catch (Exception e) {  
 System.err.println("Error: " + e.getMessage());  
 } finally {  
 scanner.close();  
 }  
}

**Startup Sequence:**

1. 1. Initialize all repositories (StudentRepository, TeacherRepository, etc.)
2. 2. Load data from JSON files (students.json, teachers.json, etc.)
3. 3. Initialize FileUploadService
4. 4. Create Scanner for user input
5. 5. Run OOP features demonstration (Inheritance, Polymorphism, Encapsulation, Generics, Exception Handling)
6. 6. Display interactive menu for user interaction
7. 7. Wait for user commands

# 6. Java OOP Concepts Implemented

This project demonstrates all major OOP concepts. Here's where each is implemented:

**1. Classes & Objects**

* • User.java, Student.java, Teacher.java, Admin.java, Principal.java
* • Course.java, Department.java, Batch.java
* • Assignment.java, Grade.java, Message.java, Submission.java
* • All domain classes represent real-world entities

**2. Encapsulation**

* • All classes: Private fields with public getters/setters
* • User.java: Private userId, name, email, username, password
* • InputValidator.java: Encapsulated validation logic
* • FileUploadService.java: Private upload directory and metadata

**3. Inheritance**

* • User.java (abstract base class)
* • Student.java extends User
* • Teacher.java extends User
* • Admin.java extends User
* • Principal.java extends User
* • BaseException.java (abstract) → All custom exceptions

**4. Polymorphism**

* • Method Overriding: getRole() in all User subclasses
* • Dynamic Method Dispatch: User[] array holding different subtypes
* • Constructor Overloading: Multiple constructors in User, Course
* • Interface implementation: Repository<T>, Searchable<T>, Sortable<T>

**5. Abstraction**

* • Abstract class: User.java (abstract getRole() method)
* • Abstract class: BaseException.java
* • Interfaces: Repository.java, Searchable.java, Sortable.java
* • Interface: UploadService.java

**6. Generics**

* • Repository<T> - works with any entity type
* • Searchable<T> - generic search interface
* • Sortable<T> - generic sort interface
* • UploadService<T> - generic upload interface
* • StudentRepository implements Repository<Student>
* • Student implements Searchable<Course>, Sortable<Course>

**7. Exception Handling**

* • Custom exception hierarchy: BaseException
* • ValidationException, NotFoundException, RepositoryException
* • AuthenticationException, AuthorizationException, UploadException
* • Try-catch-finally blocks in Main.java
* • Exception logging with timestamps

**8. Collections Framework**

* • ArrayList<Student>, ArrayList<Teacher>, ArrayList<Course>
* • HashMap<String, String> for file metadata
* • Stream API: filter(), map(), collect() operations
* • Sorting and searching with lambda expressions

**9. Access Modifiers**

* • private: All class fields (userId, name, email, etc.)
* • public: Getters, setters, main methods
* • protected: Could be used for subclass access
* • default (package-private): Some utility methods

**10. Constructors**

* • Default constructors: User(), Student(), Course()
* • Parameterized constructors with validation
* • Constructor chaining: super() in subclasses
* • Constructor overloading in Course.java

**11. Packages**

* • sms.app - Application entry point
* • sms.domain - Entity classes
* • sms.data - Repository layer
* • sms.exceptions - Custom exceptions
* • sms.services - Business logic
* • sms.search, sms.sort - Interfaces
* • sms.validation - Input validation

**12. File Handling**

* • JSON file read/write in all Repository classes
* • ObjectMapper for JSON serialization/deserialization
* • File upload handling in FileUploadService.java
* • Files: students.json, teachers.json, courses.json, etc.

# 7. Additional Services & Functionality

The project includes several specialized services beyond basic OOP:

**🔍 Search Service**

* Interface: Searchable.java (sms.search package)
* Implementation: Student.java, Teacher.java, Admin.java
* Functionality: Search courses, students, or any entity by criteria
* Uses Stream API for filtering: courses.stream().filter(...)

**📊 Sort Service**

* Interface: Sortable.java (sms.sort package)
* Implementation: Student.java, Teacher.java, Admin.java
* Functionality: Sort entities by name, ID, credits, etc.
* Uses Comparator and lambda expressions

**📤 File Upload Service**

* Interface: UploadService.java (sms.services)
* Implementation: FileUploadService.java
* Functionality: Upload files, validate, store, manage metadata
* Features: Role-based visibility, metadata tracking
* Storage: uploads/ directory, uploads.json for metadata

**🔐 Authentication Service**

* Location: User.java (login(), logout() methods)
* Exception: AuthenticationException.java
* Validates username and password
* Throws custom exceptions for invalid credentials

**✅ Validation Service**

* Class: InputValidator.java (sms.validation)
* Methods: validateName(), validateEmail(), validateUsername()
* Validates: Email format, password length, numeric IDs
* Throws: ValidationException with field details

**🗄️ Repository Service (Data Persistence)**

* Interface: Repository.java (generic)
* Implementations: StudentRepository, TeacherRepository, etc.
* Operations: CRUD (Create, Read, Update, Delete)
* File Handling: JSON read/write with Jackson ObjectMapper
* Methods: add(), update(), delete(), getAll(), find(), sort()

**💬 Messaging Service**

* Entity: Message.java
* Repository: MessageRepository.java
* Functionality: Internal communication between users
* Storage: messages.json

**📝 Assignment & Submission Service**

* Entities: Assignment.java, Submission.java
* Repositories: AssignmentRepository, SubmissionRepository
* Workflow: Teacher creates → Student submits → Teacher grades
* Storage: assignments.json, submissions.json

**📈 Grade Management Service**

* Entity: Grade.java
* Repository: GradeRepository.java
* Functionality: Track student grades per course
* Storage: grades.json

# Project Summary

* ✓ Build Tool: Maven 3.9.9
* ✓ Java Version: Java 17
* ✓ Main Dependency: Jackson Databind 2.15.2 (JSON processing)
* ✓ Build Plugins: maven-compiler-plugin, exec-maven-plugin
* ✓ Entry Point: sms.app.Main.main()
* ✓ Architecture: Layered (app, domain, data, services, exceptions)
* ✓ Total Packages: 8 (sms.app, sms.domain, sms.data, sms.exceptions, sms.services, sms.search, sms.sort, sms.validation)
* ✓ Total Classes: 38 Java files
* ✓ Data Storage: JSON files (students.json, teachers.json, courses.json, etc.)
* ✓ File Uploads: uploads/ directory with metadata in uploads.json
* ✓ OOP Concepts: All 12 major concepts fully implemented
* ✓ Design Patterns: Repository Pattern, Strategy Pattern (via interfaces)

**How to Build and Run:**

# Compile the project  
mvn clean compile  
  
# Run the application  
mvn exec:java  
  
# Or combine both  
mvn clean compile exec:java  
  
# Build JAR file  
mvn clean package  
  
# Run JAR file  
java -jar target/student-management-system-1.0.0.jar