Maven and JavaFX

Josh Wilcox (jw14g24)

March 24, 2025

Table of Contents

1. Maven
   - Project Structure
   - POM structure
   - Adding Dependencies
   - Maven Lifecycle
   - Maven Plugins
   - Documentation with Maven
   - Maven Best Practices

Maven

What is Maven?

- **Build Tool**
  - Takes code and makes it into a library
  - Makes Jar files, docstrings, API, etc
- **Project Management Tool**
  - Uses a POM File to give versioning info
  - Implements logs, docs, javadocs, code coverage etc
- **Dependency Management Tool**
  - Automatically downloads dependencies using the POM file

Project Structure

- **Standard Directory Layout:**
  - `src/main/java` - Application/Library sources
  - `src/main/resources` - Application/Library resources
  - `src/test/java` - Test sources
  - `src/test/resources` - Test resources
  - `target/` - Compiled classes and built packages (generated)
- **pom.xml** - Project Object Model
  - Defines project configuration and dependencies
  - Specifies build plugins and goals
  - Declares project metadata (name, version, etc.)
  - Manages dependency versions and conflicts

POM structure

- **Basic POM Elements:**
  - `<groupId>`, `<artifactId>`, `<version>` - Identifies the project uniquely
  - `<name>`, `<description>` - Human-readable project information
  - `<dependencies>` - External libraries required by the project
  - `<build>` - Build configuration (plugins, resources, etc.)
- **Properties Section:**
  - Allows defining custom variables for reuse throughout the POM
  - Used to centralize version numbers and configuration values
  - Example:

```
<properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <javafx.version>19</javafx.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

  - Access properties with `${property.name}` syntax throughout the POM
  - Used in dependency declarations: `<version>${javafx.version}</version>`
  - Simplifies updating multiple dependencies with the same version

Dependencies in POM

- **Dependencies Section Structure:**
  - Enclosed within `<dependencies>...</dependencies>` tags
  - Each dependency defined in its own `<dependency>` element
  - Example:

```xml
<dependencies>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-controls</artifactId>
        <version>${javafx.version}</version>
    </dependency>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-fxml</artifactId>
        <version>${javafx.version}</version>
    </dependency>
</dependencies>
```

- **Dependency Coordinates:**
  - `groupId` - Organization that created the library
  - `artifactId` - Name of the library
  - `version` - Specific version of the library
  - Optional `scope` - When the dependency is needed:
    - `compile` (default) - Available during compile, test, and runtime
    - `test` - Available only for test compilation and execution
    - `provided` - Provided by JDK or container at runtime
    - `runtime` - Not needed for compilation, only at runtime
- **Transitive Dependencies:**
  - Maven automatically resolves dependencies of dependencies
  - Simplifies dependency management - no need to list all indirect dependencies
  - Can be controlled with `<exclusions>` to prevent conflicts
- **Repository Sources:**
  - Maven Central - default public repository
  - Custom repositories can be specified in `<repositories>` section
  - Local repository (`/.m2/repository`) caches all downloaded dependencies

Adding Dependencies: Logging with Log4j

- **Why Use Logging?**
  - More structured and persistent than print statements
  - Different log levels (DEBUG, INFO, WARN, ERROR, etc.)
  - Can be configured to output to different destinations (console, file, database)
  - Can be enabled/disabled without code changes
- **Adding Log4j Dependencies:**

```xml
<dependencies>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>2.22.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.22.1</version>
    </dependency>
</dependencies>
```

- **Configuration:**
  - Requires `log4j2.xml` configuration file in `src/main/resources`
  - Defines loggers, appenders, and log levels

Log4j Example Usage

```java
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class LoggingExample {
    // Create a logger instance for this class
    private static final Logger logger = LogManager.getLogger(LoggingExample.class);

    public static void main(String[] args) {
        // Different log levels
        logger.trace("Trace message - finest level of detail");
        logger.debug("Debug message - useful for developers");
        logger.info("Info message - general information");
        logger.warn("Warning message - potential issues");
        logger.error("Error message - something went wrong");
        logger.fatal("Fatal message - severe error causing termination");

        // Logging with parameters
        String user = "John";
        logger.info("User {} logged in successfully", user);

        // Logging exceptions
        try {
            int result = 10 / 0;
        } catch (Exception e) {
            logger.error("Calculation error", e);
        }
    }
}
```

Log4j Configuration Example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
        </Console>
        <File name="File" fileName="logs/app.log">
            <PatternLayout
            ↪  pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
        </File>
    </Appenders>
    <Loggers>
        <Root level="info">
            <AppenderRef ref="Console"/>
            <AppenderRef ref="File"/>
        </Root>
    </Loggers>
</Configuration>
```

Maven Build Lifecycle

- **What is a Build Lifecycle?**
  - A well-defined sequence of phases that define the build process
  - Each phase executes a series of goals in a specific order
  - Maven has three built-in lifecycles: clean, default, and site
- **Default Lifecycle Main Phases:**
  - `validate` - Validates project structure and information
  - `compile` - Compiles source code
  - `test` - Runs tests using a test framework
  - `package` - Packages compiled code into distributable format (e.g., JAR)
  - `verify` - Runs checks to verify package validity
  - `install` - Installs package into local repository
  - `deploy` - Copies final package to remote repository
- **Running a Lifecycle Phase:**

```
mvn compile      # Compiles the code
mvn test         # Compiles and runs tests
mvn package      # Compiles, tests, and packages
mvn install      # Compiles, tests, packages, and installs locally
```

- **Key Concept:** Running any phase executes all previous phases

Maven Plugins Overview

- **What are Maven Plugins?**
  - Plugins perform tasks during the Maven build
  - They implement the actual functionality of the build process
  - Maven itself does very little - plugins do the actual work
- **Types of Plugins:**
  - **Build plugins** - Execute during the build process
    - Configured in the `<build>` section of POM
    - Example: compiler, surefire (testing), jar, exec
  - **Reporting plugins** - Generate reports and documentation
    - Configured in the `<reporting>` section of POM
    - Example: javadoc, site, project-info-reports
- **Plugin Goals:**
  - Plugins consist of ”goals” (specific tasks)
  - Example: compiler:compile, compiler:testCompile
  - Format: `plugin-prefix:goal`
- **POM Structure for Plugins:**

```
<build>
    <plugins>
        <plugin>
            <groupId>...</groupId>
            <artifactId>...</artifactId>
            <version>...</version>
            <configuration>...</configuration>
            <executions>...</executions>
        </plugin>
    </plugins>
</build>
```

Build Plugin: Maven Exec Plugin

- **Purpose:** Executes Java programs or system commands
- **Configuration:**

```xml
<build>
    <plugins>
        <!-- Exec Maven Plugin -->
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <version>3.1.0</version>
            <executions>
                <execution>
                    <goals>
                        <goal>java</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <mainClass>comp1206.App</mainClass>
            </configuration>
        </plugin>
    </plugins>
</build>
```

- **Usage:**

```
mvn exec:java                    # Runs the configured Java class
mvn exec:java -Dexec.mainClass="com.example.MainClass"   # Override main class
mvn exec:java -Dexec.args="arg1 arg2"  # Pass command-line arguments
```

- **Key Benefits:**
  - Simplifies running applications from command line
  - Ensures classpath includes all dependencies
  - Can be bound to Maven lifecycle phases

Build Plugin: Maven JAR Plugin

- **Purpose:** Creates JAR files from compiled classes
- **Configuration:**

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>3.3.0</version>
            <configuration>
                <archive>
                    <manifest>
                        <addClasspath>true</addClasspath>
                        <classpathPrefix>lib/</classpathPrefix>
                        <mainClass>com.example.MainClass</mainClass>
                    </manifest>
                </archive>
            </configuration>
        </plugin>
    </plugins>
</build>
```

- **Usage:**

```
mvn package                    # Creates JAR as part of package phase
mvn jar:jar                    # Creates JAR directly
```

- **Creating Executable JARs:**
  - The `mainClass` configuration specifies the entry point
  - `addClasspath` adds Class-Path entry to manifest
  - For self-contained JARs with dependencies, use maven-shade-plugin or maven-assembly-plugin
- **JAR Location:** `target/[artifactId]-[version].jar`

Creating Self-Contained Executable JARs

- **Problem:** Standard JAR files don't include dependencies
- **Solution 1: Maven Shade Plugin**

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>3.4.1</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>shade</goal>
            </goals>
            <configuration>
                <transformers>
                    <transformer
                    ↪ implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransfo
                        <mainClass>com.example.MainClass</mainClass>
                    </transformer>
                </transformers>
            </configuration>
        </execution>
    </executions>
</plugin>
```

- **Solution 2: Maven Assembly Plugin**

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>3.6.0</version>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
            <manifest>
                <mainClass>com.example.MainClass</mainClass>
            </manifest>
        </archive>
    </configuration>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Javadoc with Maven

- **What is Javadoc?**
  - Documentation generator for Java code
  - Creates HTML pages from Java code comments
  - Standard for Java API documentation
- **Javadoc Plugin Configuration:**

```xml
<reporting>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-javadoc-plugin</artifactId>
            <version>3.5.0</version>
            <configuration>
                <show>private</show>
                <nohelp>true</nohelp>
                <source>${maven.compiler.source}</source>
            </configuration>
        </plugin>
    </plugins>
</reporting>
```

- **Executing Javadoc:**

```
mvn javadoc:javadoc    # Generate Javadoc
mvn site               # Generate full project site including Javadoc
```

- **Javadoc Comment Format:**

```java
/**
 * This class represents a user in the system.
 *
 * @author Your Name
 * @version 1.0
 * @since 2023-01-01
 */
public class User {
    /**
     * Creates a new user with the specified name.
     *
     * @param name the name of the user
     * @throws IllegalArgumentException if name is null or empty
     * @return a new User instance
     */
    public static User createUser(String name) {
        // Implementation
    }
}
```

Maven Best Practices

- **Project Structure:**
  - Follow standard directory layout
  - Use meaningful package names
  - Place resources in appropriate directories
- **Dependencies:**
  - Use dependency versions explicitly or through properties
  - Keep dependencies up-to-date but stable
  - Use appropriate scopes (compile, test, provided, runtime)
  - Be aware of transitive dependencies and conflicts
- **Build Configuration:**
  - Use properties for centralized configuration
  - Configure plugins declaratively in POM
  - Avoid hardcoded values, use properties instead
  - Use profiles for environment-specific configurations
- **Multi-Module Projects:**
  - Break large projects into modules
  - Use parent POMs for common configuration
  - Leverage dependency management
- **Testing:**
  - Include unit and integration tests
  - Configure test plugins (surefire, failsafe)
  - Use test resources appropriately