Build Tools – maven und intellIJ

Inhaltsverzeichnis

1 Build-Tools

Ein Build-Tool ist ein Programm, das den gesamten Prozess der Softwareerstellung automatisiert – von der Übersetzung des Quellcodes (in was auch immer) über das Testen bis hin zur Paketierung und Bereitstellung lauffähiger Artefakte (z. B. JAR-, WAR- oder EAR-Dateien).

Es arbeitet i.d.R. unabhängig von der Entwicklungsumgebung und folgt einem definierten Build-Prozess, der über Skripte oder Konfigurationsdateien gesteuert wird:

- `pom.xml` bei Maven,
- `build.gradle` bei Gradle
- `Makefile` bei C/Make

Ein Build-Tool stellt damit eigentlich sicher, dass der Code auf jedem System gleich gebaut wird — automatisiert, reproduzierbar und ohne manuelle Eingriffe.

2 IDE

Eine IDE (Integrated Development Environment) ist primär ein Werkzeug zur Softwareentwicklung – sie bietet Komfortfunktionen wie Code-Editor, Syntax-Highlighting, Autovervollständigung, Debugger und grafische Projektverwaltung. Moderne IDEs wie IntelliJ IDEA, Eclipse oder VS-Code integrieren Build-Tools. Die IDEs dienen als Benutzeroberfläche, um das Build-Tool im Hintergrund auszuführen. Es dauert manchmal ein bisschen bis man sich dessen gewahr wird.

Kurz gesagt:

- Das **Build-Tool** definiert **wie** lauffähige Software gebaut wird (technischer Prozess).
- Die **IDE** unterstützt **wie** man Code schreibt (Entwicklungsprozess).

Ein Build-Tool kann vollständig ohne IDE ausgeführt werden (z. B. in CI/CD-Pipelines oder auf Servern), während eine IDE ohne Build-Tool zwar Code anzeigen, aber kein standardisiertes Artefakt erzeugen könnte.

3 Apache Maven

Apache Maven ist **das** standardisierte Build- und Projektmanagement-System für Java, das Softwareprojekte deklarativ über eine zentrale Konfigurationsdatei (pom.xml) beschreibt.

Es automatisiert den gesamten Build-Prozess – vom Kompilieren über das Testen bis hin zum Erstellen lauffähiger Artefakte wie JAR- oder WAR-Dateien.

Maven verwaltet externe Bibliotheken (*Dependencies*) zentral und lädt sie automatisch aus definierten Repositories, inklusive ihrer transitiven Abhängigkeiten. Das ist eine echte Erleichterung der Entwicklungsarbeit, der Schritt sich die letzte Version von jar Files zu organisieren und dann einzubinden entfällt einfach.

Durch seinen Lebenszyklus und das Prinzip „Konvention vor Konfiguration“ ermöglicht Maven reproduzierbare Builds unabhängig von der Entwicklungsumgebung.

3.1 pom.xml

Apache Maven beschreibt also deklarativ, ***was*** gebaut werden soll – das ***wie*** übernimmt Maven automatisch anhand standardisierter Regeln.

Im Zentrum steht die Datei `pom.xml` (*Project Object Model*). Sie enthält alle Informationen über das Projekt:

- Name
- Version
- Quellverzeichnis
- Zielplattform
- externe Abhängigkeiten (Dependencies)
- und Build-Plugins.

Diese XML-basierte Struktur ermöglicht eine konsistente Verwaltung von Projekten und vereinfacht das Teilen von Code, da sich jedes Maven-Projekt mit einem einzigen Befehl bauen lässt.

3.2 Build-Lifecycle

Maven folgt einem festen **Build-Lifecycle**, der in Phasen unterteilt ist:

→ `validate`
→ `compile`
→ `test`
→ `package`
→ `verify`
→ `install`
→ `deploy`

Jede Phase ruft standardisierte **Plugins** auf (z. B. zum Kompilieren, Testen oder Packen in ein JAR/WAR). Damit sind alle Maven-Projekte konsistent aufgebaut und reproduzierbar – unabhängig von der Entwicklungsumgebung.

3.3 Dependency Management.

Ein zentraler Aspekt ist das **Dependency Management**.

Statt Bibliotheken manuell als JAR-Dateien einzubinden, werden sie mit einer Deklaration in der `pom.xml`. angegeben. Maven lädt die benötigten Versionen automatisch aus einem Repository wie Maven Central oder einem firmeneigenen Artefaktserver herunter – inklusive ihrer transitiven Abhängigkeiten (also der Abhängigkeiten der Abhängigkeiten).

Dadurch bleibt das Projekt-Setup schlank und nachvollziehbar: Alle externen Bibliotheken sind über ihre Koordinaten (Group-ID, Artifact-ID, Version) eindeutig definiert. Zudem wird die Verwaltung von Versionen, Upgrades und Konflikten (Dependency Mediation) zentralisiert und automatisiert.

Immer mehr Softwareprojekte gehen dazu über, kein *.jar File mehr zu verteilen, sondern sich auf Maven Central zu stützen.

3.4 IDE-Integration

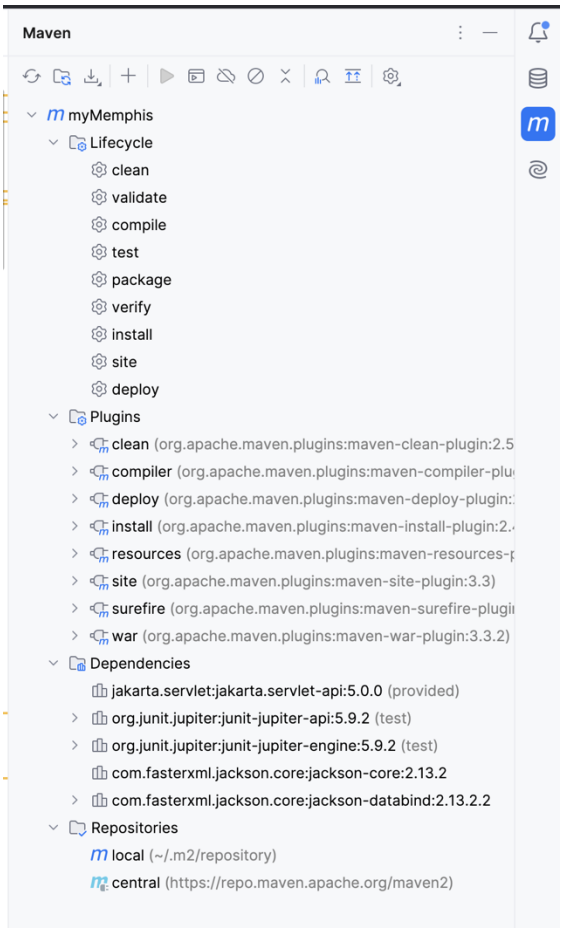
IDE-Integrationen wie IntelliJ IDEA oder Eclipse können Maven-Projekte direkt importieren und erkennen sofort Struktur, JDK, Abhängigkeiten und Build-Prozess – was manuelle Konfiguration weitgehend überflüssig macht.

Damit ist Maven sowohl ein Werkzeug zur Automatisierung als auch ein formales Projekt-modell, das Struktur, Wiederverwendbarkeit und Nachvollziehbarkeit in der Java-Entwicklung sicherstellt.

4 Arbeiten mit Maven in IntelliJ IDE

IntelliJ IDEA bietet eine vollständige Integration von Maven-Projekten. Beim Öffnen eines Projekts, das eine pom.xml enthält, erkennt IntelliJ automatisch die Maven-Struktur und importiert alle Abhängigkeiten aus dem lokalen Repository (`~/.m2/repository`).

Die IDE erzeugt daraufhin den Klassenpfad, markiert Quell- und Ressourcenordner korrekt (z.B. `src/main/java`, `src/main/resources`) und aktiviert die Build-Ziele über das integrierte **Maven-Toolfenster**.



Über dieses Toolfenster lassen sich die wichtigsten Build-Phasen direkt ausführen:

- **compile** – kompiliert den Quellcode,
- **test** – führt Unit-Tests aus,
- **package** – erstellt das Artefakt (z. B. eine JAR-Datei),
- **install** – installiert das Artefakt im lokalen Repository.

IntelliJ kann Maven-Befehle auch automatisch bei Änderungen an der pom.xml neu ausführen (Auto-Import).

Darüber hinaus ermöglicht eine *Run Configuration*, Programmargumente zu setzen und eine `main`-Klasse direkt zu starten – Maven kümmert sich dabei weiterhin im Hintergrund um das Dependency Management.

Man kann aber maven einfach im Hintergrund arbeiten lassen, und die IDE als GUI nutzen um die einzelnen Build Phasen auszuführen.

5 Eine pom.xml

Und jetzt mal zum eigentlichen Kern – der pom.xml. Datei. Das folgende Listing zeigt die Maven-Konfiguration des Projekts *F1ExcPrev*. Dieses „Projekt“ liest Rennergebnisse aus einer Excel-Datei und verwendet dafür Apache POI (zur Excel-Verarbeitung) und PostgreSQL (für Zugriff un das Speichern in einer postgres Datenbank).

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

```
  <groupId>com.formulaone</groupId>
  <artifactId>F1ExcPrev</artifactId>
  <version>1.0-SNAPSHOT</version>

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

  <dependencies>
    <!-- Apache POI Core-Bibliothek (XLS-Unterstützung) -->
    <dependency>
      <groupId>org.apache.poi</groupId>
      <artifactId>poi</artifactId>
      <version>5.2.5</version>
    </dependency>

    <!-- Apache POI OOXML-Bibliothek (XLSX-Unterstützung) -->
    <dependency>
      <groupId>org.apache.poi</groupId>
      <artifactId>poi-ooxml</artifactId>
      <version>5.2.5</version>
    </dependency>

    <!-- PostgreSQL JDBC-Treiber -->
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>42.7.3</version>
    </dependency>
  </dependencies>
</project>
```

5.1 Erläuterung der zentralen Elemente

Abschnitt	Bedeutung
<groupId>	Eindeutige Kennung der Organisation oder des Projekts. Hier: com.formulaone
<artifactId>	Name des erzeugten Artefakts (z. B. JAR-Datei). Hier: F1ExcPrev
<version>	Projektversion. -SNAPSHOT kennzeichnet eine laufende Entwicklungsversion.
<properties>	Globale Eigenschaften, z. B. Java-Version (source/target) und Encoding.
<dependencies>	Liste aller externen Bibliotheken, die Maven automatisch lädt.

5.2 Verwendete Abhängigkeiten

- **Apache POI (poi, poi-ooxml)**
  Bibliothek zur Verarbeitung von Excel-Dateien im HSSF- (XLS) und XSSF- (XLSX) Format. Wird hier benutzt, um Rennergebnisse aus Excel-Dateien auszulesen und zu parsen.
- **PostgreSQL JDBC-Treiber**
  Stellt die JDBC-Schnittstelle bereit, um aus Java heraus SQL-Query an eine PostgreSQL-Datenbank zu senden.

6 Ausführen eines Maven-Projekts ohne IDE

Ein Maven-Projekt lässt sich vollständig über die Kommandozeile steuern. Voraussetzung ist eine installierte JDK-Umgebung und Maven selbst (Befehl `mvn` im Systempfad).

Typischer Ablauf:

1. **In das Projektverzeichnis wechseln** (dort, wo die `pom.xml` liegt)

   ```
   cd C:\Users\<Name>\IdeaProjects\F1ExPrev
   ```

2. **Build starten**

   ```
   mvn clean package
   ```

   Kompiliert die Java-Quellen unter_ `src/main/java`
   Kopiert die Ressourcen aus : `src/main/resources`
   Lädt alle Abhängigkeiten aus aus dem Maven-Repository
   Erstellt das JAR-Artefakt im Ordner `target/`

3. **Ergebnis prüfen**

   Nach erfolgreichem Build liegt das erzeugte Artefakt z. B. unter:

   ```
   target/F1ExPrev-1.0-SNAPSHOT.jar
   ```

4. **Programm ausführen**

   Der Start erfolgt mit dem Java-Interpreter, wobei der Klassenpfad (`-cp`) auf das erzeugte JAR verweist und die Hauptklasse angegeben wird:

   ```
   java -cp target/F1ExcelPreviewer-1.0-SNAPSHOT.jar
       com.formulaone.F1ExcPrev
   ```

   Falls eine „Fat JAR“ (*jar-with-dependencies*) erstellt wurde, kann sie direkt gestartet werden:

   ```
   java -jar target/F1ExPre-1.0-SNAPSHOT-jar-with-dependencies.jar
   ```

5. **Optional:**

   Wenn zusätzliche Parameter übergeben werden (z. B. der Pfad zu einer Excel-Datei), kann dies einfach an den Java-Aufruf angehängt werden:

   ```
   Java -cp target/F1ExcelPreviewer-1.0-SNAPSHOT.jar
      com.formulaone.F1ExcelPreviewer  "C:\data\race_results.xlsx"
   ```

Durch diesen Ansatz bleibt das Projekt **plattformunabhängig** und kann auf jedem System mit Java und Maven gebaut und ausgeführt werden – unabhängig von IntelliJ IDE.