What is Maven?

Maven is a **build automation** and **dependency management tool** for Java projects. It uses a configuration file called `pom.xml`.

Core Functions of Maven

1. **Build Management**: Recreate builds for any environment.
2. **Dependency Management**: Automatically download Java libraries (JARs) from remote repositories.
3. **Repository Use**:
    - `.m2/repository`: Local repository
    - **Remote repo**: If dependency not in local, Maven fetches from central/remote repo

Maven Directory Hierarchy

- Local → Remote → Central repo
- Path: `C:/Users/<user>/.m2/repository/...`
- If `.m2` doesn't contain a required dependency, it fetches from central repo.

Common Maven Commands & Lifecycle Phases

Phase	Description
`mvn clean`	Deletes the `target/` directory (cleans workspace)

`mvn compile`	Compiles source code and generates `.class` files into `target/`
`mvn test`	Executes unit tests using JUnit/TestNG
`mvn package`	Creates a `.jar`/`.war`/`.ear` package
`mvn install`	Installs built package to local repo (`~/.m2/repository`)
`mvn deploy`	Deploys built artifact to remote repo (like Nexus/Artifactory)

mvn clean compile

mvn clean install

mvn deploy

Types of Build Artifacts

- `.jar`: Java ARchive (classes only)
- `.war`: Web ARchive (web applications)
- `.ear`: Enterprise ARchive (combined apps)

pom.xml (Project Object Model)

Defines the structure and configuration of a Maven project.

Key Tags:

xml

CopyEdit

```
<project>

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.companyname.project</groupId>

  <artifactId>myproject</artifactId>

  <version>1.0</version>

  <packaging>jar</packaging>



  <dependencies>

    <!-- Add required library dependencies here -->

  </dependencies>

</project>
```

How to Create a Simple Maven Project

1. Create folder
2. Create Java file (`HelloWorld.java`)
3. Create `pom.xml`

4. Run Maven goals like:

```
mvn clean install
```

Maven Setup Notes

- Maven needs:
    - Java installed (`java -version`)
    - Maven installed (`mvn -version`)
- In Linux:

```
sudo apt install maven
```

Difference Between `Fork` and `Clone`

Feature	Fork	Clone
Definition	Makes a **copy of a repository under your GitHub account**	Creates a **local copy** of a repository on your machine
Where?	Happens on **GitHub/GitLab UI** (remote server)	Happens on your **local machine** via command line
Purpose	To contribute to someone else's repo (without access)	To work with a repo locally

Use Case	Open-source collaboration, PRs to upstream repo	Day-to-day development and editing
Command	Done via GitHub/GitLab website → “Fork” button	`git clone <repo-url>`
Creates	A separate copy in your GitHub account	A working directory with `.git` metadata on your machine
Link to original	Keeps link to original repo (for pull requests)	No direct GitHub fork link (only remote origin)

Typical Workflow Comparison

When You Fork

1. You click **Fork** on GitHub repo.
2. It creates a **copy in your account**.
3. You then `git clone` **your forked repo**.
4. Make changes locally, push to your fork, and raise a **pull request** to the original repo.

When You Clone

1. You run `git clone <repo-url>`.
2. Work locally, commit, and push back to the **same remote**, if you have access.
3. No need to fork unless you're contributing to a third-party repo without permission.

Summary

- **Fork**: Remote → GitHub account (used for contribution without permission)
- **Clone**: Remote → Local machine (used for local development)

Maven Repository Types

Type	Description
Local Repository	Your own machine's cache of dependencies (located in `.m2/repository`)
Remote Repository	Repositories hosted on remote servers (e.g., internal Nexus/Artifactory)
Central Repository	The default public repository provided by Maven (https://repo.maven.apache.org)

1. Local Repository

- Created **automatically** by Maven on your machine.

Location:

`Windows: C:\Users\<username>\.m2\repository`

- `Linux/Mac: ~/.m2/repository`

When you run:

```
mvn install
```

- it stores artifacts (JARs, POMs) in this location.

✅ **Purpose**: Avoid downloading dependencies repeatedly.

2. Remote Repository

- Hosted by your **organization or project team**.
- Examples:
  - **Nexus Repository Manager**
  - **JFrog Artifactory**
- Contains **custom or internal JARs** not found in the public Maven Central.

Defined in `pom.xml`:

```
<repositories>
  <repository>
    <id>company-repo</id>
    <url>https://nexus.company.com/repository/maven-releases/</url>
  </repository>
</repositories>
```

-

✅ **Use case**: For enterprise/private builds or hosting internal libraries.

3. Central Repository

- Publicly hosted by the Maven project:
  👉 https://repo.maven.apache.org/maven2
- Contains most open-source Java libraries (Spring, JUnit, etc.)

✅ **Default fallback** when:

- Dependency is not found in local `.m2`
- No custom remote repo is defined

Maven Download Order (Dependency Resolution)

When Maven needs a dependency:

1. **Checks local repo** (`.m2/repository`)
2. If **not found**, checks configured **remote repos**
3. If still not found, downloads from **Maven Central**
4. Saves it into **local repo** for future use

Summary Table

Repo Type	Location	Used For
Local	`.m2/repository` on your computer	Caching downloaded artifacts
Remote	Internal repo like Nexus	Company-specific or private JARs

Central	`https://repo.maven.apache.org`	Open-source dependencies

What are Dependencies in Maven?

In Maven, **dependencies** are **external Java libraries** (JAR files) that your project needs to compile, run, or test.

For example:

- `JUnit` for unit testing
- `Spring Boot` libraries
- `Apache Commons` utilities

Instead of manually downloading and adding these JARs, Maven fetches them **automatically** based on the configuration inside the `pom.xml` file.

Where Are Dependencies Downloaded?

1. **First, Maven checks the local repository:**

Path:

```
C:\Users\<your-username>\.m2\repository\
```

- ○
- ○ This is called the **local Maven repo**.

Example folder:

```
.m2\repository\junit\junit\4.13.2\junit-4.13.2.jar
```

- ○

2. **If not found locally**, Maven goes to the **central remote repository**:
    - URL: https://repo.maven.apache.org/maven2/
3. **It downloads the required JAR files** (and any of their dependencies!) and saves them in the `.m2` local repo folder.

How Are Dependencies Defined?

You define dependencies inside your `pom.xml`:

xml

CopyEdit

```
<dependencies>

  <dependency>

    <groupId>junit</groupId>

    <artifactId>junit</artifactId>

    <version>4.13.2</version>

    <scope>test</scope>

  </dependency>

</dependencies>
```

This tells Maven:

- Group: `junit`
- Artifact: `junit`
- Version: `4.13.2`
- Scope: `test` (used only during testing phase)

How Maven Resolves Dependencies

When you run:

```
mvn compile
```

or

```
mvn install
```

Maven performs:

1. **Checks** `.m2/repository` if the dependency exists.
2. If **not found**, downloads from the **central repository**.
3. **Stores it locally** in `.m2` so next builds are faster.
4. If the dependency has **transitive dependencies** (its own dependencies), Maven downloads them too.

Summary Table

Step	**Action**
1️⃣	Check `.m2` local repo
2️⃣	If missing, download from Maven Central

3️⃣ Store in `.m2\repository\`

4️⃣ Make JARs available to your project during compile/test/run

What is `pom.xml`?

`pom.xml` stands for **Project Object Model**.
It is the **heart of a Maven project**. This file tells Maven everything it needs to know to build your project.

It contains configuration and metadata such as:

- Project info (name, version, etc.)
- Dependencies (external libraries like JUnit, Spring, etc.)
- Build plugins
- Repository details
- Java version compatibility
- Packaging type (e.g., `jar`, `war`, `ear`)

Basic Structure of `pom.xml`

xml

CopyEdit

```
<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.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

</project>
```

Tag	Description
`<groupId>`	Unique ID for your project (usually domain style, e.g., `com.example`)
`<artifactId>`	Name of the project (e.g., `myapp`, `orderservice`)
`<version>`	Version of the project (e.g., `1.0.0`, `1.0-SNAPSHOT`)
`<packaging>`	Type of artifact to build: `jar`, `war`, `pom`, etc.
`<dependencies>`	Lists all external libraries your project needs

`<build>`	Optional: includes plugins or build configurations

Why `pom.xml` is Important for DevOps

As a DevOps engineer, you must understand `pom.xml` because:

- It automates the build process (`compile`, `test`, `package`, `install`, `deploy`)
- It ensures **consistent builds** across environments (CI/CD)
- It's used by tools like **Jenkins**, **GitHub Actions**, **Azure DevOps**, etc., to run Maven tasks
- It declares and pulls dependencies automatically, avoiding manual JAR downloads

Dependency Management Example

xml

CopyEdit

```xml
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.7.5</version>
  </dependency>
</dependencies>
```

Maven will download the **Spring Boot Web Starter** and its **transitive dependencies** and make them available for your project.

Here's a **sample Java project** you can compile using Maven. It includes:

- A simple Java class
- A test class
- A `pom.xml` file to configure the build

```
sample-maven-project/
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── com
    │           └── example
    │               └── App.java
    └── test
        └── java
            └── com
                └── example
                    └── AppTest.java
```

```
<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.example</groupId>
<artifactId>sample-maven-project</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>

<name>Sample Maven Project</name>

<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
  <!-- JUnit for testing -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

```
</project>


package com.example;


public class App {
    public static void main(String[] args) {
        System.out.println("Hello, Maven!");
    }


    public int add(int a, int b) {
        return a + b;
    }
}


package com.example;


import org.junit.Test;
import static org.junit.Assert.assertEquals;


public class AppTest {


    @Test
    public void testAdd() {
        App app = new App();
        assertEquals(5, app.add(2, 3));
    }
}
```

How to Compile & Run

Open terminal and navigate to the project root:

```
cd sample-maven-project
```

1. Compile the project:

   ```
   mvn clean compile
   ```
2. Run the tests:

   ```
   mvn test
   ```
3. Package into a JAR:

   ```
   mvn package
   ```
4. Run the application:

   ```
   java -cp target/sample-maven-project-1.0-SNAPSHOT.jar
   com.example.App
   ```

Step-by-Step: How to Write pom.xml

🧾 Basic pom.xml Template

xml

CopyEdit

```
<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">

  <!-- POM model version -->
  <modelVersion>4.0.0</modelVersion>

  <!-- Project coordinates -->
  <groupId>com.mycompany</groupId>
  <artifactId>myapp</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>

  <!-- Optional metadata -->
  <name>My Sample Maven Project</name>
  <description>This is a simple Java Maven project</description>
  <url>http://www.mycompany.com/myapp</url>

  <!-- Dependencies -->
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
```

```xml
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <!-- Optional: Build plugins -->
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>

</project>
```

📘 What Each Section Means

Tag	Description
`<modelVersion>`	Always 4.0.0 (required)
`<groupId>`	Your organization or domain (e.g., com.mycompany)
`<artifactId>`	Project name
`<version>`	Version of your project (1.0, 1.0-SNAPSHOT, etc.)
`<packaging>`	jar, war, pom, etc. (defaults to jar)
`<dependencies>`	External libraries (JARs) your project uses
`<build><plugins>`	Optional section to define compiler or packaging plugins

Example: Create a Sample Maven Java Project

Create project folder

```
mkdir myapp

cd myapp
```

1. **Create `src` directory**

   ```
   mkdir -p src/main/java/com/mycompany
   ```
2. **Add Java class** `src/main/java/com/mycompany/HelloWorld.java`

```
     package com.mycompany;

public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello from Maven!");

    }

}
```

3. **Add `pom.xml` (in project root)** → Use the template above

Build the project

```
mvn clean compile

mvn package
```

✅ Output:

- Compiled `.class` files in `target/classes`

- Packaged `.jar` file in `target/`