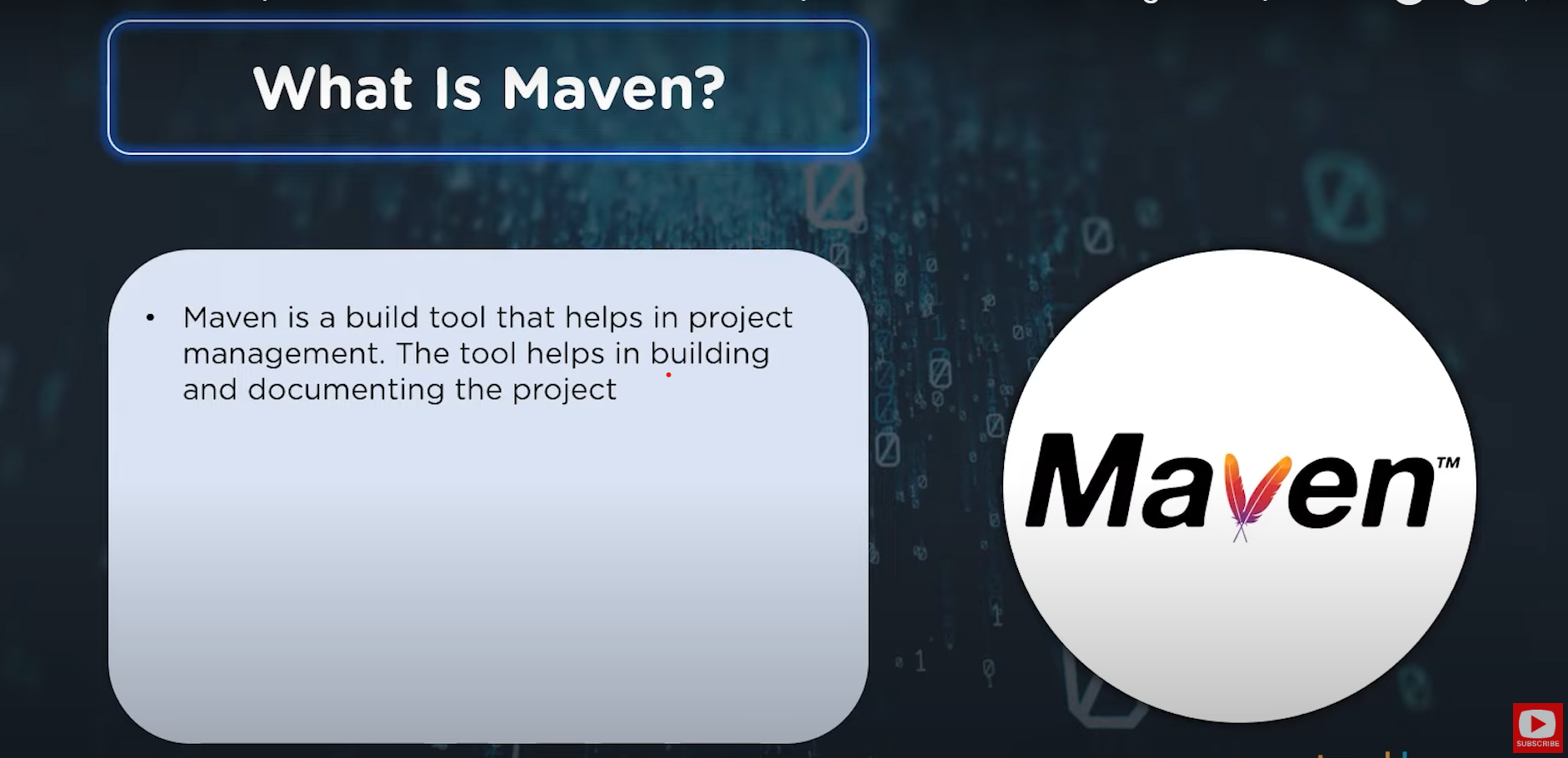
**Maven**

****

**A screenshot of a computer

Description automatically generated**

**What is Apache Maven?**

Maven is a **build automation tool** used primarily for Java projects. It helps manage a project's build, reporting, and documentation from a central piece of information. Think of it as a project management tool that simplifies the process of building and managing any Java-based project.

**Key Concepts**

1. **POM (Project Object Model)**: The core of Maven is the POM file (pom.xml), which contains information about the project and configuration details used by Maven to build the project. Example of a simple pom.xml:

<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>my-app</artifactId>

<version>1.0-SNAPSHOT</version>

</project>

1. **Dependencies**: Maven manages project dependencies (libraries or other projects your project depends on). You specify these in the pom.xml, and Maven downloads them from a central repository. Example of adding a dependency:

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

1. **Repositories**: Maven uses repositories to store and retrieve dependencies. There are local repositories (on your machine) and remote repositories (like Maven Central).
2. **Plugins**: Plugins are used to perform tasks such as compiling code, running tests, and packaging the code into a JAR file. Maven has a wide range of plugins for different tasks.

**How Maven Works**

1. **Build Lifecycle**: Maven follows a lifecycle to build and deploy projects. The default lifecycle includes phases like validate, compile, test, package, verify, install, and deploy.
2. **Goals**: Each phase is made up of goals. For example, the compile phase has a goal to compile the source code.

**Example Workflow**

1. **Creating a Project**: You can create a new Maven project using the command:

mvn archetype:generate -DgroupId=com.example -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

1. **Building the Project**: To compile the project, run:

mvn compile

1. **Running Tests**: To run tests, use:

mvn test

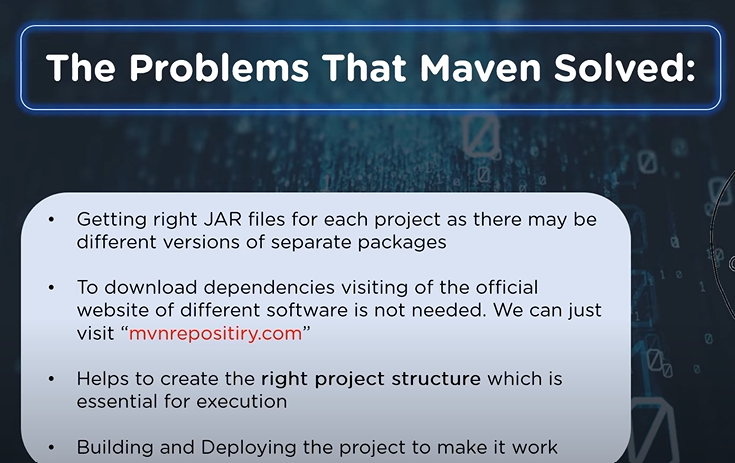
1. **Packaging the Project**: To package the project into a JAR file, run: sh mvn package

**JAR Files**

A **JAR (Java ARchive) file** is a package file format used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file for distribution. Essentially, it's a way to bundle all the components of a Java application into a single file. **Importance of JAR Files in Maven**

In Maven, JAR files play a crucial role for several reasons:

1. **Packaging**: Maven uses JAR files to package the compiled Java classes and resources into a single, distributable file. This makes it easier to share and deploy Java applications. Example: When you run the mvn package command, Maven compiles your code and packages it into a JAR file located in the target directory.
2. **Dependency Management**: Maven manages project dependencies, which are often JAR files. These dependencies are specified in the pom.xml file, and Maven automatically downloads them from a central repository and includes them in your project.
3. **Build Process**: Maven's build lifecycle includes phases like compile, test, and package. The package phase is responsible for creating the JAR file, which is the final output of the build process.



A screenshot of a computer

Description automatically generated

**Maven Lifecycle**

* **Goals: Goals** are the smallest units of work in Maven. They are specific tasks that Maven can perform. **Example**: The goal compiler:compile compiles the source code of your project. Here, compiler is the plugin name, and compile is the goal within that plugin.
* **Phases: Phases** are steps in the build lifecycle. Each phase can execute one or more goals. **Example**: The compile phase in Maven will execute the compiler:compile goal to compile your project’s source code.
* **Lifecycles: Lifecycles** are collections of phases that define the sequence of steps to build and deploy your project. Maven has three built-in lifecycles: default, clean, and site.
* **Default Lifecycle**: This is the main lifecycle that handles project deployment. It includes phases like validate, compile, test, package, verify, install, and deploy.
* **Clean Lifecycle**: This lifecycle handles project cleaning, with phases like pre-clean, clean, and post-clean.
* **Site Lifecycle**: This lifecycle handles the creation of your project’s site documentation, with phases like pre-site, site, and post-site.

**How They Work Together**

When you run a Maven command, you typically specify a phase. Maven will execute that phase and all the preceding phases in the lifecycle.

* **Example**: If you run mvn package, Maven will execute the following phases in the default lifecycle:
  1. validate: Check if the project is correct and all necessary information is available.
  2. compile: Compile the source code.
  3. test: Run tests to ensure the code is correct.
  4. package: Package the compiled code into a JAR file.

**Maven’s Standard Directory Structure**

Maven follows a convention over configuration approach, which means it uses sensible defaults for where it expects to find certain files in your project. This makes it easier to manage and build projects without needing a lot of custom configuration.

**Key Directories in a Maven Project**

1. **Source Code Directory**: This is where you put your main Java source code files.
   * **Location**: src/main/java
   * **Example**: If you have a Java class called MyApp.java, it should be placed in src/main/java.
2. **Resources Directory**: This is where you put resource files like property files, configuration files, and other non-Java files that your application needs.
   * **Location**: src/main/resources
   * **Example**: If you have a configuration file called config.properties, it should be placed in src/main/resources.
3. **Test Source Code Directory**: This is where you put your Java unit test files.
   * **Location**: src/test/java
   * **Example**: If you have a test class called MyAppTest.java, it should be placed in src/test/java.
4. **Test Resources Directory**: This is where you put resource files needed for testing, like test configuration files.
   * **Location**: src/test/resources
   * **Example**: If you have a test configuration file called test-config.properties, it should be placed in src/test/resources.
5. **Web Application Directory**: This is where you put web application files like HTML, JSP, and web configuration files if you are building a web application.
   * **Location**: src/main/webapp
   * **Example**: If you have an HTML file called index.html, it should be placed in src/main/webapp.

**Why Use These Standard Locations?**

* **Consistency**: By following these standard locations, your project structure will be consistent with other Maven projects. This makes it easier for others to understand and work with your project.
* **Less Configuration**: Maven knows where to look for files by default, so you don’t need to configure these locations manually. This reduces the amount of configuration you need to write.

**Preserving folders within Git**

**Why Do We Need to Preserve Folders?**

Git, by default, only tracks files, not empty folders. This means if you create a folder structure for your project but don’t put any files in those folders, Git won’t save those empty folders. However, sometimes you want to keep these folders in your repository, even if they are empty, so that the structure is maintained when someone else checks out your project.

**How to Preserve Folders Using**.gitkeep

To make Git track these empty folders, we can create a small, empty file inside each folder. A common practice is to use a file named .gitkeep.

* **Commands**:
  + touch src/main/java/.gitkeep (and similar for other folders)
  + git status to check the status
  + git add . to stage the files
  + git commit -m "Adding .gitkeep files to preserve folder structure" to commit the changes

**Maven Local Repository**

1. **Understanding the**install**Command**: The install command in Maven is used to compile your code, run tests, package your application into a JAR file, and then copy that JAR file into your local Maven repository. This local repository is a special folder on your computer where Maven stores all the libraries and artifacts it needs.
2. **Running the**install**Command**: Open your terminal and navigate to your project directory. Run the following command: mvn clean install
   * Here’s what this command does:
     + clean: Deletes the target directory to start fresh.
     + install: Compiles the code, runs tests, packages the application, and copies the JAR file to the local repository.
3. **Target Directory**: Contains all the files generated during the build process, including the final JAR file.
4. **Local Maven Repository**: A special folder on your computer where Maven stores all the libraries and artifacts it needs. Typically located in the .m2 folder within your home directory.
5. **Navigating the Repository**: Organized by group ID, artifact ID, and version number, making it easy to find your project’s artifacts.

**Plugins**

**What Are Maven Plugins?**

Maven plugins are tools that add specific capabilities to your Maven build process. They perform tasks such as compiling code, running tests, packaging applications, and more. Almost all functionality in Maven is provided by plugins. Think of Maven as the engine, and plugins as the tools that do the actual work.

**Key Points About Maven Plugins**

1. **Purpose**:
   * Plugins extend Maven’s core functionality. They are essential for performing various build tasks.
   * Example tasks include compiling Java code, running unit tests, generating documentation, and deploying applications.
2. **Types of Plugins**

* **Core Plugins:** These are built into Maven and provide basic functionality.
* **Third-Party Plugins:** These are additional plugins that you can add to your project to extend Maven’s capabilities**.**

**Commonly Used Plugins**

* **Compiler Plugin:** Compiles your Java source code.
* **Surefire Plugin:** Runs unit tests**.**
* **JAR Plugin:** Packages your compiled code into a JAR file.
* **Deploy Plugin:** Deploys your project to a remote server.
* **Site Plugin**: Generates project documentation.

1. **How Plugins Work**:
   * Plugins are configured in the pom.xml file.
   * Each plugin can have multiple goals. A goal is a specific task that the plugin performs.
   * Example: The Compiler Plugin has goals like compile and testCompile.
2. **Changing Plugin Configuration:** Sometimes, you need to change the default behavior of a plugin. For example, the default Java version for the Compiler Plugin might be 1.5, but you want to use Java 8. You can change this in the pom.xml

**Dependency**

**What Are Dependencies in Maven?**

Dependencies in Maven are external libraries or modules that your project needs to work. Instead of manually downloading and adding these libraries to your project, Maven handles this for you. You simply declare the dependencies in your pom.xml file, and Maven downloads them from a central repository.

**Why Are Dependencies Important?**

Dependencies are crucial because they allow you to use existing libraries and frameworks in your project, saving you time and effort. They also ensure that your project has all the necessary components to compile and run correctly.

**How Dependencies Are Managed**

1. **Maven Central Repository**: By default, Maven downloads dependencies from the Maven Central Repository, an online repository containing a vast collection of libraries.
2. **Local Repository**: Maven first checks your local repository (a cache on your computer) before downloading from the central repository. This speeds up the build process and reduces the need for repeated downloads.

**How to Declare Dependencies**

Dependencies are declared in the pom.xml file within the <dependencies> section. Each dependency is specified with a <dependency> element that includes the group ID, artifact ID, and version of the library.

**Explanation of the Example**

* **groupId**: This is the group or organization that created the library. For JUnit, it’s junit.
* **artifactId**: This is the name of the library. For JUnit, it’s junit.
* **version**: This specifies the version of the library you want to use. For JUnit, it’s 4.12.
* **scope**: This defines the classpath for the dependency. In this case, test means the dependency is only needed for testing.

**How Maven Manages Dependencies**

1. **Downloading Dependencies**:
   * When you run a Maven command like mvn compile or mvn install, Maven checks the pom.xml file for dependencies.
   * Maven then downloads these dependencies from a central repository (like Maven Central) and stores them in your local repository (usually located in the .m2 directory in your home folder).
2. **Transitive Dependencies**:
   * Maven also handles transitive dependencies. This means if your project depends on Library A, and Library A depends on Library B, Maven will automatically download both Library A and Library B.
   * This ensures that all necessary libraries are available for your project.

**Managing Dependency Versions**

Sometimes, different dependencies might require different versions of the same library. Maven allows you to manage these versions to avoid conflicts.

**Dependency Scopes**

Maven supports six dependency scopes, which determine when and how the dependencies are used:

1. **Compile** (default): Needed for compiling and running the project.
2. **Runtime**: Needed only at runtime, not for compilation.
3. **Test**: Needed only for testing, not for compilation or runtime.
4. **Provided**: Provided by the runtime environment (e.g., application server).
5. **System**: Local dependencies specified with an absolute path (rarely used).
6. **Import**: Used to import dependencies from other projects (special cases).

**Trying to Compile Without the Dependency**

**Objective**: Attempt to compile the project without adding the necessary dependency to see what happens.

1. **Open the Project**: Navigate to the root of your project directory in the terminal.
2. **Check the Java File**: Open the Application.java file. This file references the StringUtils class from the apache.commons.lang3 library, but we haven’t added this library to our project yet.
3. **Attempt to Compile**:
   * Run the following command to clean and compile the project:
   * mvn clean compile
   * **Expected Result**: The compilation will fail with an error message saying that the package org.apache.commons.lang3 does not exist. This is because Maven doesn’t know about the commons-lang3 library yet.

**Listing All Dependencies:** Get a list of all dependencies used in the project.

**Use the Dependency Plugin**: Run the following command to list all dependencies:

**mvn dependency:tree**

* + This command will download the dependency plugin if it’s not already cached and then print a tree of all dependencies and their sub-dependencies.

**Unit Testing**

**What is Unit Testing?**

Unit testing is a way to test individual parts (units) of your code to ensure they work correctly. Each unit test checks a small piece of functionality in isolation from the rest of the application. This helps catch bugs early and makes your code more reliable.

**Why Use Unit Testing?**

* **Catch Bugs Early**: Find and fix issues before they become bigger problems.
* **Ensure Code Quality**: Verify that your code behaves as expected.
* **Facilitate Refactoring**: Make changes to your code with confidence, knowing that tests will catch any errors.
* **Documentation**: Tests can serve as documentation for how your code is supposed to work.

**How to Do Unit Testing in Maven**

Maven uses the Surefire Plugin to run unit tests. By default, Maven looks for test classes in the src/test/java directory and runs any methods annotated with @Test.

**Setting up and Running Unit tests in Maven**

* 1. **Preparing the Project for Unit Testing**
* **Create the Test Directory Structure:** Navigate to your project directory in the terminal.

Create the necessary directories for your test files. Maven expects test files to be in src/test/java

* **Add a Sample Test File**
* **Save and Close the File**
  1. **Adding JUnit Dependency:** Add the JUnit library to your project for unit testing.
* **Find the JUnit Dependency:** Go to Maven Central and search for junit. Find the latest version of JUnit (e.g., 4.12) and note the groupId, artifactId, and version.
* **Modify the pom.xml File:** Open the pom.xml file in your text editor. Add the JUnit dependency within the <dependencies> section
* Save and Close the pom.xml file
  1. **Run the Tests**
* In your terminal, run the following command to execute the tests: **mvn test**
* Maven will compile the test classes and run all methods annotated with @Test.
* **Check Test Results:** After running the tests, Maven will display the results in the terminal. You can see which tests passed and which failed.
  1. **Viewing Test Reports**: Check the surefire-reports directory for detailed test reports. You will find two types of files**: .txt and .xml.**

Open the .txt file to see a human-readable summary of the test results: **cat Application.txt**

This file shows the number of tests run, failures, errors, and skipped tests.

**Maven within Eclipse**

**Step-by-Step Guide**

**1. Creating a New Maven Project in Eclipse:** Set up a new Maven project in Eclipse.

1. **Open Eclipse**: Launch Eclipse from your dock (Mac) or desktop icon (Windows).
2. **Start a New Maven Project**: In Eclipse, go to the Project Explorer, right-click, select **New**, then **Other**. Scroll down to **Maven**, select **Maven Project**, and click **Next**.
3. **Choose Project Location**: Uncheck **Use default Workspace location**. Click **Browse** and navigate to your desired project directory (e.g., a projects folder in your home directory).
4. **Select an Archetype**: Maven provides archetypes as templates for new projects. The default **maven-archetype-quickstart** is a good starting point for a simple Java project.
5. **Enter Project Details**:
   * **Group Id**: A unique identifier for your project, usually in reverse domain name format (e.g., clinic.programming.training).
   * **Artifact Id**: The name of your project (e.g., eclipse-example).
   * **Version**: Leave as 0.0.1-SNAPSHOT.
   * **Package**: This will be filled automatically based on the Group Id but can be adjusted.
6. **Review Project Structure**: Right-click the project, select **Properties**, and check the location to ensure it’s in your specified directory. Expand the project in the Project Explorer to see the folder structure: src/main/java, src/test/java, and pom.xml.
7. **Edit pom.xml**: Double-click pom.xml to open it. Eclipse provides a graphical interface for editing, but you can also view and edit the raw XML by clicking the pom.xml tab.

**2. Importing an Existing Maven Project into Eclipse:** Import an existing Maven project into Eclipse.

1. **Open Eclipse**: Ensure Eclipse is open with your default workspace.
2. **Import the Project**:
   * Right-click in the Project Explorer, select **Import**, then **Import…**.
   * Expand **Maven** and select **Existing Maven Projects**, then click **Next**.
3. **Select Project Directory**:
   * Click **Browse** and navigate to the directory containing your Maven project (e.g., projects folder).
   * Select the project directory and click **Open**.
   * Eclipse will detect the Maven projects in the folder. Ensure the desired project is checked and click **Finish**.

**3. Reviewing and Editing pom.xml in Eclipse** Understand and modify the pom.xml file in Eclipse.

1. **Open pom.xml**: Expand your project in the Project Explorer and double-click pom.xml.
2. **Overview Tab**: View general project information, including Group Id, Artifact Id, and Version.
3. **Dependencies Tab**: See a list of project dependencies. You can add, remove, or modify dependencies here.
4. **Dependency Hierarchy**: View all dependencies and their transitive dependencies in a hierarchical format.
5. **Effective POM**: See the combined POM, including inherited configurations from parent POMs and the super POM.
6. **Raw XML**: Click the pom.xml tab to view and edit the raw XML directly.

**Archetype**

**What Are Maven Archetypes?**

Maven archetypes are project templates that help you quickly set up a new project with a predefined structure and configuration. Think of them as blueprints for creating different types of projects, such as a simple Java application, a web application, or a library.

**Why Use Maven Archetypes?**

* **Consistency**: Ensure that all projects start with a consistent structure.
* **Speed**: Quickly set up new projects without having to manually create directories and configuration files.
* **Best Practices**: Use templates that follow best practices for project setup.

**How to Use Maven Archetypes**

**Creating a New Project with an Archetype**

1. **Open Your Terminal**: Navigate to the directory where you want to create your new project.
2. **Run the Maven Archetype Command**:
   * Use the following command to create a new project with the Maven Quickstart archetype:

**mvn archetype:generate -DgroupId=com.example -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false**

* + **Explanation**:
    - **mvn archetype:generate**: Tells Maven to generate a new project using an archetype.
    - **-DgroupId=com.example**:  A unique identifier for your project, usually in reverse domain name format (e.g., com.example). Type it and press Enter.
    - **-DartifactId=my-app**: The name of your project (e.g., quick-start-example). Type it and press Enter.
    - **-DarchetypeArtifactId=maven-archetype-quickstart**: Specifies the archetype to use (in this case, the Quickstart archetype).
    - **-DinteractiveMode=false**: Runs the command without prompting for additional input.

1. **Project Structure**: After running the command, Maven creates a new project with the following structure:

my-app

├── pom.xml

└── src

├── main

│ └── java

│ └── com

│ └── example

│ └── App.java

└── test

└── java

└── com

└── example

└── AppTest.java

**Custom Archetypes:** You can also create your own custom archetypes if you have specific project templates you want to reuse. This involves defining the structure and files of the template and packaging it as a Maven archetype.

* **Exploring the Project**: Review the pom.xml file and the source directories to understand the project structure.
* **Building the Project**: Use mvn clean install to compile, test, and package the project.
* **Cleaning Up**: Optionally, remove the project if you no longer need it.