



Dependency Managers. Maven.

Contenido

Dependency Managers. Maven.	1
1. Dependency Managers.....	3
2. Java Dependency Managers	4
2.1. Maven.....	4
2.2. Gradle	4
3. Deep inside Maven.....	5
3.1. Convention over Configuration.....	5
3.2. Maven Lifecycle.....	6
3.3. The POM file.....	7
3.4. Creating a Maven Project with IntelliJ IDEA.....	10
4. Bibliography.....	14

1. Dependency Managers

A dependency is an external standalone program module (library) that can be as small as a single file or as large as a collection of files and folders organized into packages that performs a specific task. For example, if you are writing a database application, you will need to install the appropriate driver (ODBC or JDBC) in order to be able to connect to the database. In other words, your application is dependent on the package (driver) for connecting to the database.

There are several ways in which you may face a dependency issue:

- Resolving dependencies during installation: Some types of applications do not automatically search/download for program dependencies. In these cases, you need to install them manually.
- Lack of dependencies after an uninstall: On certain occasions, when you remove software that makes use of a shared library, the library is also uninstalled, causing the operating system or the application that made use of it to malfunction.
- Version issues: In some cases, different programs make use of the same library, but need different versions. Therefore, different versions of the same library should coexist, causing the client applications to malfunction.

Dependency managers are software modules that coordinate the integration of external libraries or packages into larger application stack. Dependency managers use configuration files like `composer.json`, `package.json`, `build.gradle` or `pom.xml` to determine:

1. What dependency to get
2. What version of the dependency in particular and
3. Which repository to get them from.

A repository is a source where the declared dependencies can be fetched from using the name and version of that dependency. Most dependency managers have dedicated repositories where they fetch the declared dependencies from. For example, maven central for maven and gradle, npm for npm, and packagist for composer.

So when you declare a dependency in your config file — e.g. `composer.json`, the manager will goto the repository to fetch the dependency that match the exact criteria you have set in the config file and make it available in your execution environment for use.

Some example of Dependency Managers are:

1. Composer (used with php projects)
2. Gradle (used with Java projects including android apps. and also is a build tool)
3. Node Package Manager (NPM: used with NodeJS projects)
4. Yarn
5. Maven (used with Java projects including android apps. and also is a build tool)

2. Java Dependency Managers

The most popular dependency managers for Java are Maven and Gradle. You can use both of them from IntelliJ IDEA. In this course we will use Maven but, if you feel more comfortable using Gradle, you may (although no support to issues with Gradle will be provided).

2.1. Maven

Maven is a free software tool for managing Java projects released in 2002 by the Apache Software Foundation. It is based on xml files and comes with defined objectives to perform certain tasks such as compiling the code and building the packaged project, but Maven is also responsible for obtaining all the dependencies of the project and uploading it to the final repository so that other projects that depend on it can use it.



In addition, Maven can be used with a lot of additions provided by the developers community, for example, there are plugins that can be used for functions such as deploying the project on a server, pass the project to a code quality analyzer, or build the application test coverage.

2.2. Gradle

Gradle is also a free software tool released in 2007 but is used to manage Java, Groovy or Scala projects, although it already supports other languages. This tool is configured with Groovy-based DSL files, instead of the xml used by Maven.



Gradle has emerged to solve the problem of building management, dependencies and deployment of projects that have a language different than Java, although it also supports Java. Day by day the use of this technology is growing and as Maven, it has a large number of plugins that can be added for a multitude of tasks related to the construction, dependency management and deployment.

3. Deep inside Maven

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. Using maven we can build and manage any Java based project. This tutorial will teach you how to use Maven in your day-to-day life of any project development using Java.

3.1. Convention over Configuration

Maven uses Convention over Configuration, which means developers are not required to create build process themselves.

Developers do not have to mention each and every configuration detail. Maven provides sensible default behavior for projects. When a Maven project is created, Maven creates default project structure. Developer is only required to place files accordingly and he/she need not to define any configuration in pom.xml.

As an example, following table shows the default values for project source code files, resource files and other configurations. Assuming, `${basedir}` denotes the project location –

Item	Default
source code	<code>\${basedir}/src/main/java</code>
Resources	<code>\${basedir}/src/main/resources</code>
Tests	<code>\${basedir}/src/test</code>
Complied byte code	<code>\${basedir}/target</code>

distributable JAR	<code>\${basedir}/target/classes</code>
--------------------------	-----------------------------------------

In order to build the project, Maven provides developers with options to mention life-cycle goals and project dependencies (that rely on Maven plugin capabilities and on its default conventions). Much of the project management and build related tasks are maintained by Maven plugins.

Developers can build any given Maven project without the need to understand how the individual plugins work. We will discuss Maven Plugins in detail in the later chapters.

3.2. Maven Lifecycle

A Build Lifecycle is a well-defined sequence of phases, which define the order in which the goals are to be executed. Here phase represents a stage in life cycle. As an example, a typical **Maven Build Lifecycle** consists of the following sequence of phases:

Phase	Handles	Description
prepare-resources	resource copying	Resource copying can be customized in this phase.
validate	Validating the information	Validates if the project is correct and if all necessary information is available.
compile	compilation	Source code compilation is done in this phase.
Test	Testing	Tests the compiled source code suitable for testing framework.
package	packaging	This phase creates the JAR/WAR package as mentioned in the packaging in POM.xml.
install	installation	This phase installs the package in local/remote maven repository.
Deploy	Deploying	Copies the final package to the remote repository.

There are always pre and post phases to register goals, which must run prior to, or after a particular phase.

When Maven starts building a project, it steps through a defined sequence of phases and executes goals, which are registered with each phase.

Maven has the following three standard lifecycles:

- Clean
- Build
- Site

A goal represents a specific task which contributes to the building and managing of a project. It may be bound to zero or more build phases. A goal not bound to any build phase could be executed outside of the build lifecycle by direct invocation.

The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked. For example, consider the command below. The clean and package arguments are build phases while the dependency:copy-dependencies is a goal.

```
mvn clean dependency:copy-dependencies package
```

Here the clean phase will be executed first, followed by the dependency:copy-dependencies goal, and finally package phase will be executed.

3.3. The POM file

POM stands for Project Object Model. It is fundamental unit of work in Maven. It is an XML file that resides in the base directory of the project as pom.xml.

The POM contains information about the project and various configuration details used by Maven to build the project(s).

POM also contains the goals and plugins. While executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, and then executes the goal. Some of the configuration that can be specified in the POM are the following:

- project dependencies
- plugins
- goals
- build profiles
- project version
- developers
- mailing list

Before creating a POM, we should first decide the project group (groupId), its name (artifactId) and its version as these attributes help in uniquely identifying the project in repository.

Here you can see a POM file example:

```
<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.jrgs</groupId>
  <artifactId>U3_Maven_Example</artifactId>
  <version>1.0-SNAPSHOT</version>

  <name>U3_Maven_Example_01</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -
->
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>42.2.19</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.commons/commons-
dbcp2 -->
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-dbcp2</artifactId>
      <version>2.8.0</version>
    </dependency>
  </dependencies>
```



```

<build>
  <pluginManagement><!-- lock down plugins versions to avoid using
Maven defaults (may be moved to parent pom) -->
    <plugins>
      <!--          clean          lifecycle,          see
https://maven.apache.org/ref/current/maven-
core/lifecycles.html#clean_Lifecycle -->
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <version>3.1.0</version>
      </plugin>
      <!--          default          lifecycle,          jar          packaging:          see
https://maven.apache.org/ref/current/maven-core/default-
bindings.html#Plugin_bindings_for_jar_packaging -->
      <plugin>
        <artifactId>maven-resources-plugin</artifactId>
        <version>3.0.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
      </plugin>
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.22.1</version>
      </plugin>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.0.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-install-plugin</artifactId>
        <version>2.5.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-deploy-plugin</artifactId>
        <version>2.8.2</version>
      </plugin>
      <!--          site          lifecycle,          see
https://maven.apache.org/ref/current/maven-
core/lifecycles.html#site_Lifecycle -->
      <plugin>
        <artifactId>maven-site-plugin</artifactId>
        <version>3.7.1</version>
      </plugin>
      <plugin>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <version>3.0.0</version>
      </plugin>

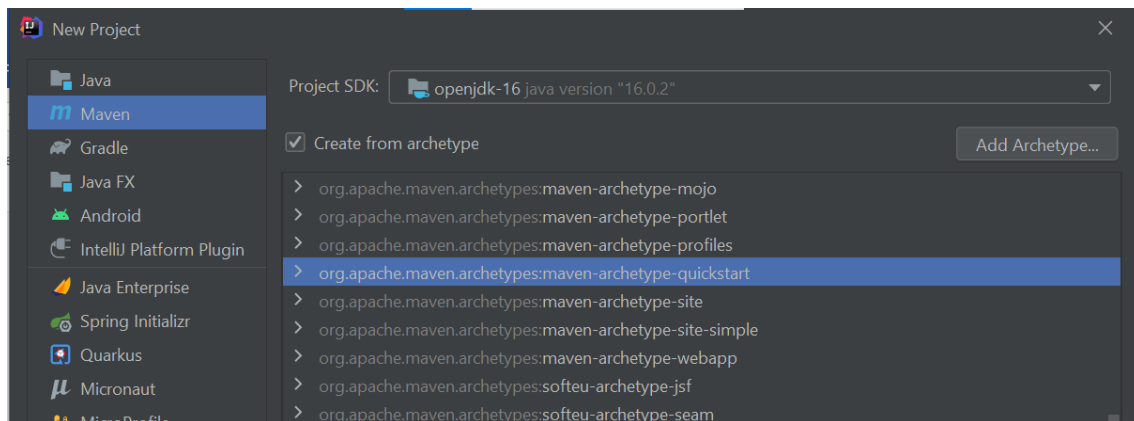
```

```
</plugins>
</pluginManagement>
</build>
</project>
```

You can learn more about POM files and Maven in the following link:
<https://www.tutorialspoint.com/maven/index.htm>

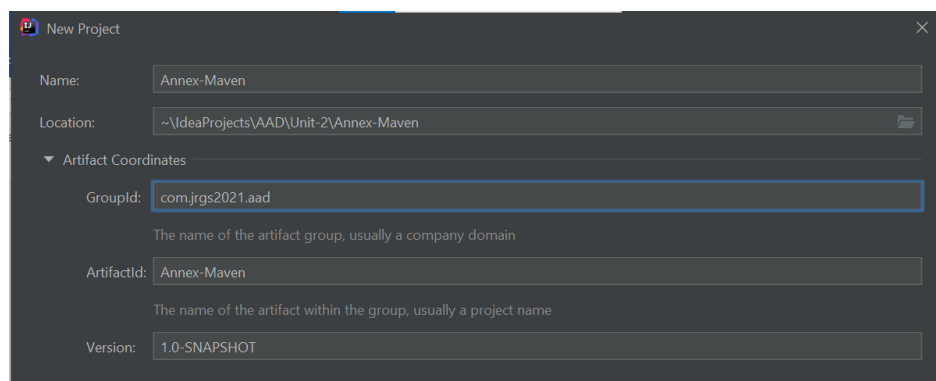
3.4. Creating a Maven Project with IntelliJ IDEA

Either you are using the community or the ultimate version of IntelliJ IDEA, you are able to create a Maven project from inside of this IDE. First of all, create a new project and select the option 'Maven Project':

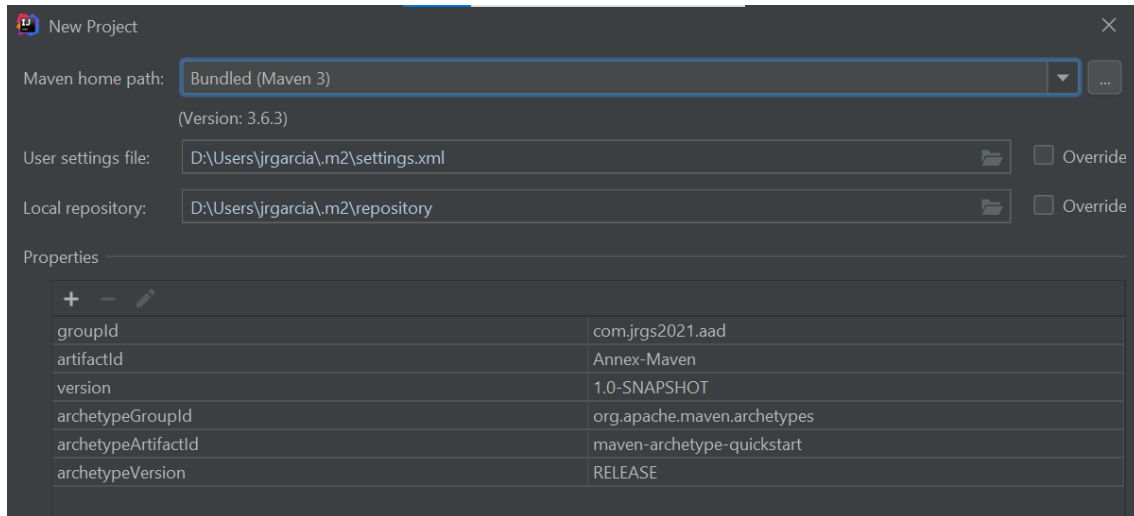


You must select an SDK (if the desired JDK is not available, you can download it in this step) and an archetype for your maven project (for the purpose of this document we will choose the quickstart archetype, but there are many more available).

In the next step we will name our project, choose a path for it and pick a groupId (the group of projects to include this one) and the artifactId (usually, the name of the project).



Finally, we will confirm the previous information, add or modify any desired property:

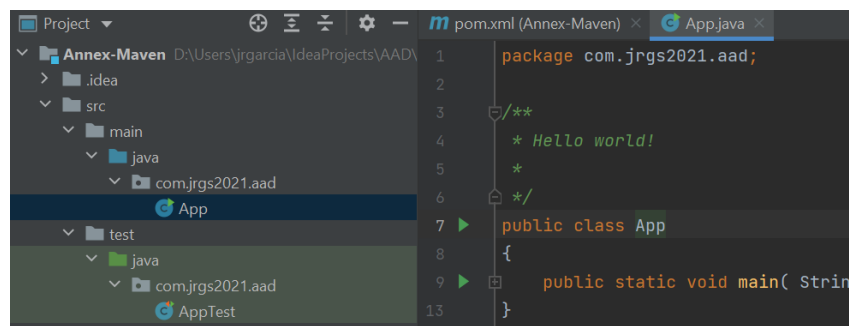


After finishing, IntelliJ IDEA will present us the POM file and will start to download any necessary plugins and indexes for our project. The first time we create a Maven project of a particular archetype this step will take a while, but the next time this step will be faster.

If we take a look to the newly created POM file, we will see that there is a dependency section that looks like the following one:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

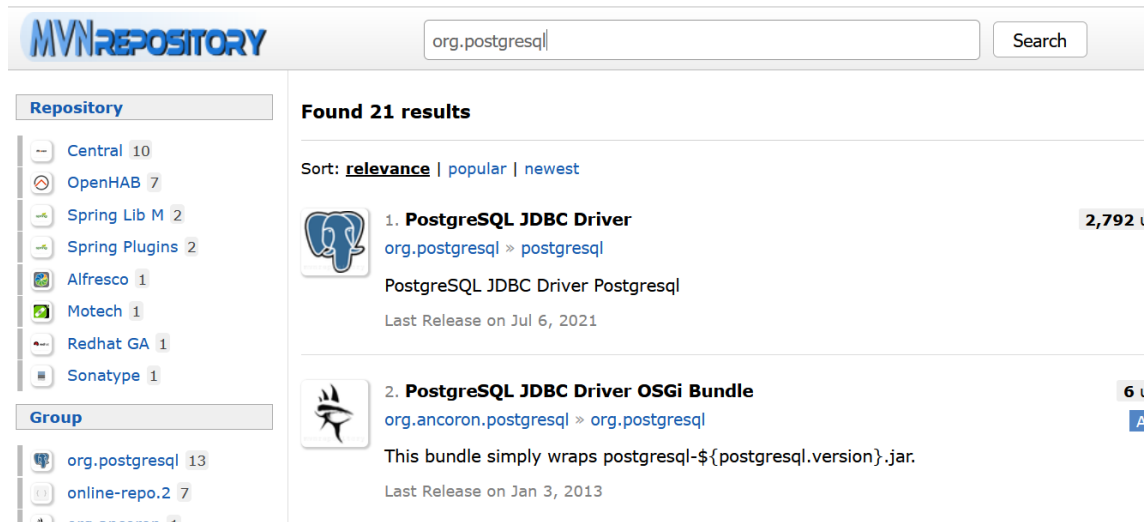
This section is notifying Maven that our project is going to use the unit testing facility available in Java, Junit. Indeed, if we inspect the code created by Maven, we will see two folders, one for the source code (main) and another one for the test code (test). Inside every folder you will find also a default class for a Java console application and a simple junit java class, too.



If we need to add a dependency to our project (for example, the use of the JDBC PostgreSQL connector) we must find the correct dependency, copy it and paste it in our POM file. The best place to search for the correct dependency is in the maven official repository:

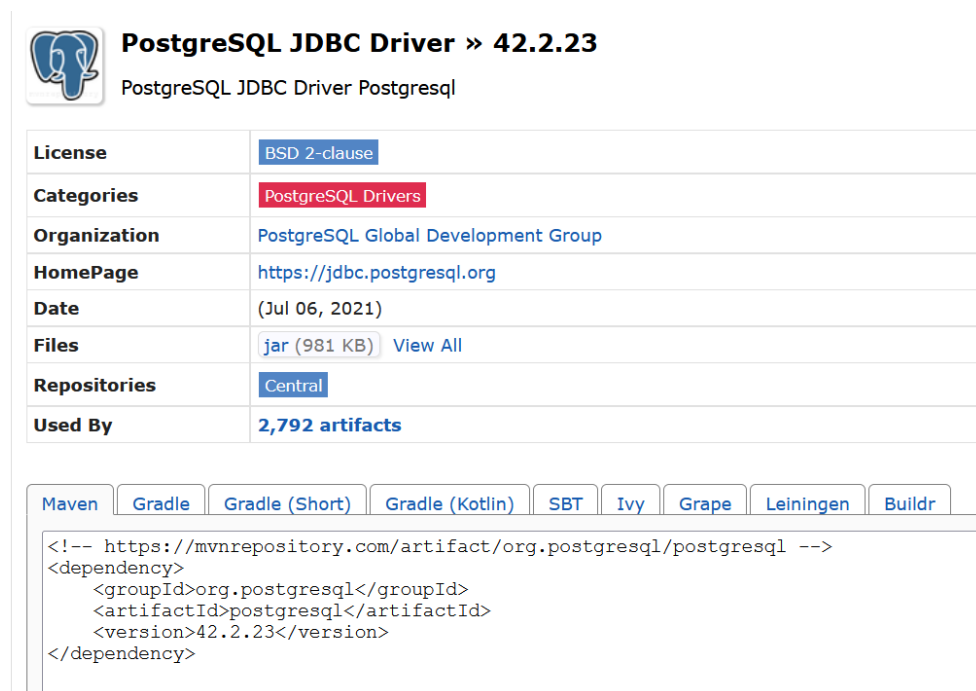
<https://mvnrepository.com>

Just type in the dependency you are looking for and you will be presented all the possible matches:



The screenshot shows the Maven Repository search interface. The search bar contains 'org.postgresql' and the search button is labeled 'Search'. On the left, there is a sidebar with 'Repository' and 'Group' filters. The 'Repository' filter shows 'Central' with 10 results, 'OpenHAB' with 7, 'Spring Lib M' with 2, 'Spring Plugins' with 2, 'Alfresco' with 1, 'Motech' with 1, 'Redhat GA' with 1, and 'Sonatype' with 1. The 'Group' filter shows 'org.postgresql' with 13 results, 'online-repo.2' with 7, and 'org.springframework' with 1. The main area displays 'Found 21 results' and a sort dropdown set to 'relevance'. Two results are visible: 1. 'PostgreSQL JDBC Driver' by 'org.postgresql' with 2,792 artifacts, last released on Jul 6, 2021. 2. 'PostgreSQL JDBC Driver OSGi Bundle' by 'org.ancoron.postgresql' with 6 artifacts, last released on Jan 3, 2013.

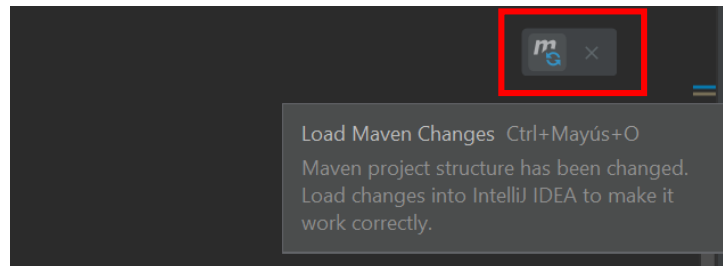
Inside every match, you will find all the versions available for that dependency. Unless we are told something different, we will choose the more recent non-beta version:



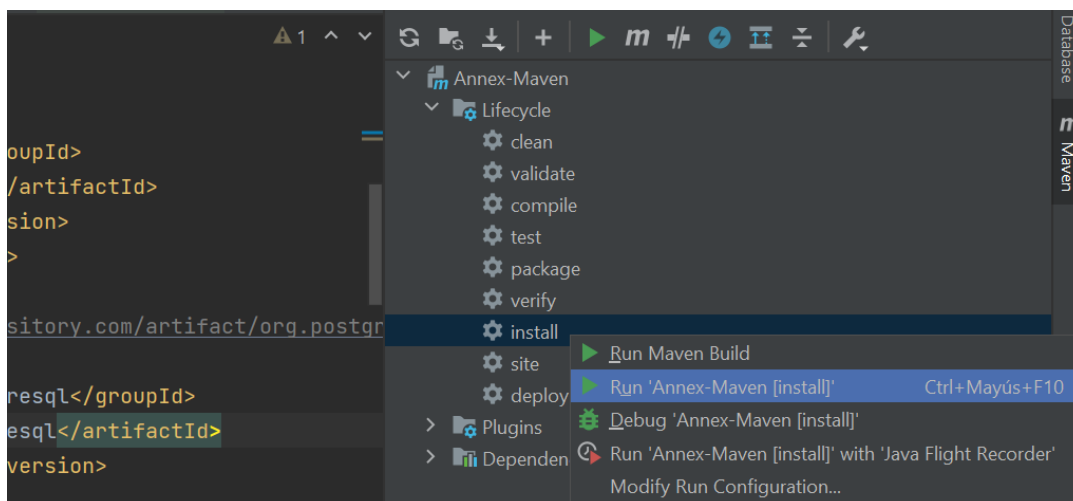
The screenshot shows the details for the 'PostgreSQL JDBC Driver' artifact, version 42.2.23. The artifact is provided by 'org.postgresql' and is available in the 'Central' repository. The license is 'BSD 2-clause'. The categories are 'PostgreSQL Drivers'. The organization is 'PostgreSQL Global Development Group'. The homepage is 'https://jdbc.postgresql.org'. The date is 'Jul 06, 2021'. The files section shows a 'jar' file (981 KB) and a 'View All' link. The 'Used By' section shows '2,792 artifacts'. Below the details, there are tabs for different build systems: Maven, Gradle, Gradle (Short), Gradle (Kotlin), SBT, Ivy, Grape, Leiningen, and Buildr. The Maven tab is selected, showing the following XML snippet:

```
<!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.2.23</version>
</dependency>
```

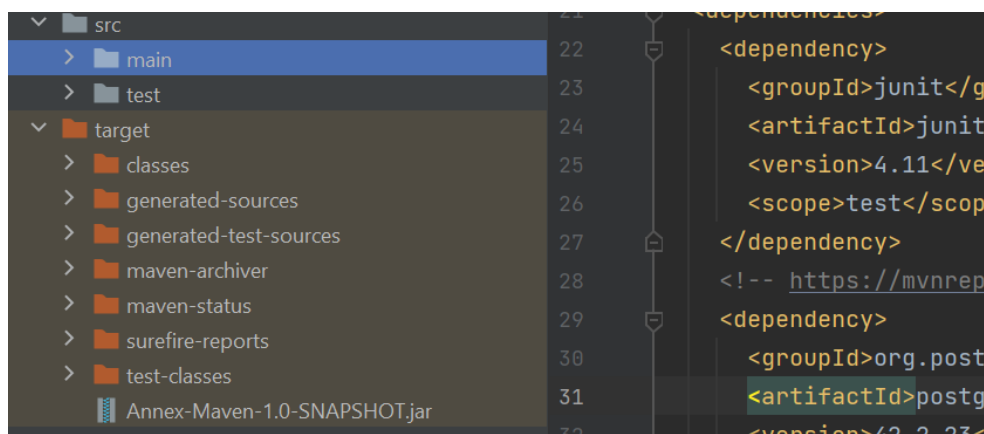
The final step is quite simple, just copy the dependency and paste it into your POM file (in the <dependencies> section). You will see the text inside the dependency marked in red color, that means that the dependency is not still available. Put the mouse over the POM file, press the right button and choose the option 'Maven/Reload project'. You can also press the icon showing that the POM file has been changed:



In IntelliJ IDEA, any of the phases of the Maven Lifecycle can be independently invoked using the right-hand side menu. For example, we can invoke the install phase to obtain an installable version of our software (in Java, a .jar file).



Once executed, we will see a new folder in our project (target) with our brand new jar file:



4. Bibliography

<https://maven.apache.org/>

<https://www.tutorialspoint.com/maven/index.htm>

<https://www.jetbrains.com/help/idea/maven-support.html>

https://www.tutorialspoint.com/maven/maven_intellij_idea.htm

<https://mvnrepository.com>