**Maven – Local, Remote and Central Repositories**

Maven repositories are physical directories which contain packaged JAR files along with extra meta data about these jar files. This meta data is in form of POM files which have jar file project information, including what other external dependencies this JAR file has. These other external dependencies are downloaded transitively into your project and become part of **effective pom** for the project.

Table of Contents

[Local repository](https://howtodoinjava.com/maven/local-remote-central-repositories/#local-repo)

[Central repository](https://howtodoinjava.com/maven/local-remote-central-repositories/#central-repo)

[Remote repository](https://howtodoinjava.com/maven/local-remote-central-repositories/#remote-repo)

**Local repository**

**Maven local repository** reside in the developer’s machine. Whenever you run maven goals which require these dependencies, maven will download the dependencies from remote servers and store them into developer’s machine.

By default, Maven create the local repository inside user home directory i.e. C:/Users/superdev/.m2 directory. You can change the location of the local repository in [setting.xml](https://howtodoinjava.com/maven/maven-settings-file/) file using localRepository tag.

|  |
| --- |
| <settings>      <localRepository>          C:\M2      </localRepository>  </settings> |

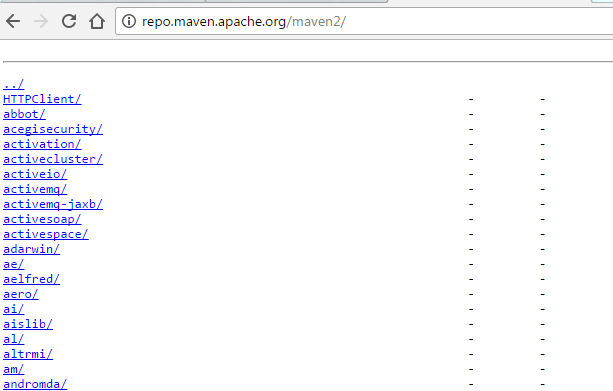
Having stored the dependencies into local machine has two main benefits. First, multiple projects can access same artifact so it reduces the storage need. Second, as dependency is downloaded only once, it reduces the network usage as well.

5.4M

What to bring to a JOB Interview Interview Tips

**Central repository**

**Maven central repository** is located at [http://repo.maven.apache.org/maven2/](https://repo.maven.apache.org/maven2/). Whenever you run build job, maven first try to find dependency from local repository. If it is not there, then, by default, maven will trigger the download from this central repository location.

Maven Central Repository

To override this default location, you can make changes to your settings.xml file to use one or more mirrors.

You do not need to make any special configuration to allow access the central repository, except [network proxy settings](https://howtodoinjava.com/maven/maven-proxy-settings/) if you are behind any firewall.

**Remote repository**

Apart from central repository, you may have needed artifacts deployed on other remote locations. For example, in your corporate office there may be projects or modules specific to organization only. In this cases, organization can create remote repository and deploy these **private artifacts**. This remote repository will be accessible only inside organization.

These **maven remote repository** work exactly same way as maven’s central repository. Whenever an artifact is needed from these repositories, it is first downloaded to developer’s local repository and then it is used.

You can configure a remote repository in the POM file or super POM file in remote repository itself.

|  |
| --- |
| <repositories>     <repository>         <id>org.source.repo</id>         <url>http://maven.orgName.com/maven2/</url>     </repository>  </repositories> |

Drop me your questions in comments section.

How to include custom library into maven local repository?

1. Kaptcha

For example, kaptcha, a popular third party Java library, which is generating captcha image to stop spam, but it’s not available in the Maven center repository.

**Update**  
Now, the kaptcha is available in the Maven Central Repository. It is fine, even it is in the Maven Central, we still can install the Kaptcha JAR file manually into our Maven local repository.

20.9K0 Cordless Stick Vacuum, 25kpa Powerful Suction Pet Hair : Good Tech Cheap

2. mvn install

Download the [kaptcha](https://code.google.com/archive/p/kaptcha/downloads), extract it and copy the kaptcha-${version}.jar to somewhere else, for example, C drive, and run this command:

$ mvn install:install-file -Dfile=c:\kaptcha-{version}.jar -DgroupId=com.google.code

-DartifactId=kaptcha -Dversion={version} -Dpackaging=jar

For example:

> mvn install:install-file -Dfile=c:\kaptcha-2.3.jar -DgroupId=com.google.code

-DartifactId=kaptcha -Dversion=2.3 -Dpackaging=jar

[INFO] Scanning for projects...

[INFO] Searching repository for plugin with prefix: 'install'.

[INFO] ------------------------------------------------------------------------

[INFO] Building Maven Default Project

[INFO] task-segment: [install:install-file] (aggregator-style)

[INFO] ------------------------------------------------------------------------

[INFO] [install:install-file]

[INFO] Installing c:\kaptcha-2.3.jar to

D:\maven\_repo\com\google\code\kaptcha\2.3\kaptcha-2.3.jar

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESSFUL

[INFO] ------------------------------------------------------------------------

[INFO] Total time: < 1 second

[INFO] Finished at: Tue May 12 13:41:42 SGT 2009

[INFO] Final Memory: 3M/6M

[INFO] ------------------------------------------------------------------------

Done, the kaptcha jar is installed (copied) into our Maven local repository.

3. pom.xml

After installed, just declares the kaptcha dependency as usual.

pom.xml

<dependency>

<groupId>com.google.code</groupId>

<artifactId>kaptcha</artifactId>

<version>2.3</version>

</dependency>

4. Alternative Solution

Forget about the ‘mvn install`, we also can download the .jar and tell the project to find the .jar in the system path like this:

pom.xml

<dependency>

<groupId>com.google.code</groupId>

<artifactId>kaptcha</artifactId>

<version>2.3</version>

<scope>system</scope>

<systemPath>d:/projects/kaptcha.jar</systemPath>

</dependency>

pom.xml

<dependency>

<groupId>com.google.code</groupId>

<artifactId>kaptcha</artifactId>

<version>2.3</version>

<scope>system</scope>

<systemPath>${project.basedir}/lib/kaptcha.jar</systemPath>

</dependency>

# The settings.xml File in Maven

## 1. Overview

While using Maven, we keep most of the project-specific configuration in the pom.xml.

Maven provides a settings file, settings.xml, which allows us to specify which local and remote repositories it will use. We can also use it to store settings that we don't want in our source code, such as credentials.

In this tutorial, we'll learn how to use the settings.xml. We'll look at proxies, mirroring, and profiles. We'll also discuss how to determine the current settings that apply to our project.

## 2. Configurations

The settings.xml file configures a [Maven](https://www.baeldung.com/maven) installation. It's similar to a pom.xml file, but is defined globally or per user.

Let's explore the elements we can configure in the settings.xml file. The main settings element of the settings.xml file can contain nine possible predefined child elements:

<**settings** xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/settings-1.0.0.xsd">

<**localRepository**/>

<**interactiveMode**/>

<**offline**/>

<**pluginGroups**/>

<**servers**/>

<**mirrors**/>

<**proxies**/>

<**profiles**/>

<**activeProfiles**/>

</**settings**>

### 2.1. Simple Values

Some of the top-level configuration elements contain simple values:

<**settings** xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/settings-1.0.0.xsd">

<**localRepository**>${user.home}/.m2/repository</**localRepository**>

<**interactiveMode**>true</**interactiveMode**>

<**offline**>false</**offline**>

</**settings**>

The localRepository element points to the path of the system’s local repository. The [**local repository**](https://www.baeldung.com/maven-local-repository)**is where all the dependencies from our projects gets cached**. The default is to use the user's home directory. However, we could change it to allow all logged-in users to build from a common local repository.

The interactiveMode flag defines if we allow Maven to interact with the user asking for input. This flag defaults to true.

The offline flag determines if the build system may operate in offline mode. This defaults to false; however, we can switch it to true in cases where the build servers cannot connect to a remote repository.

### 2.2. Plugin Groups

The pluginGroups element contains a list of child elements that specify a groupId. A groupId is the unique identifier of the organization that created a specific Maven artifact:

<**settings** xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/settings-1.0.0.xsd">

<**pluginGroups**>

<**pluginGroup**>org.apache.tomcat.maven</**pluginGroup**>

</**pluginGroups**>

</**settings**>

**Maven searches the list of plugin groups when a plugin is used without a groupId provided** **at the command line**. The list contains the groups org.apache.maven.plugins and org.codehaus.mojo by default.

The settings.xml file defined above allows us to execute truncated Tomcat plugin commands:

mvn tomcat7:help

mvn tomcat7:deploy

mvn tomcat7:run

### 2.3. Proxies

We can configure a proxy for some or all of Maven's HTTP requests. The *proxies*element allows a list of child proxy elements, but **only one proxy can be active at a time**:

<**settings** xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/settings-1.0.0.xsd">

<**proxies**>

<**proxy**>

<**id**>baeldung-proxy</**id**>

<**active**>true</**active**>

<**protocol**>http</**protocol**>

<**host**>baeldung.proxy.com</**host**>

<**port**>8080</**port**>

<**username**>demo-user</**username**>

<**password**>dummy-password</**password**>

<**nonProxyHosts**>\*.baeldung.com|\*.apache.org</**nonProxyHosts**>

</**proxy**>

</**proxies**>

</**settings**>

We define the currently active proxy via the active flag. Then with the nonProxyHosts element, we specify which hosts are not proxied. The delimiter used depends on the specific proxy server. The most common delimiters are pipe and comma.

### 2.4. Mirrors

Repositories can be declared inside a project pom.xml. This means that the developers sharing the project code get the right repository settings out of the box.

We can use mirrors in cases where we want to **define** **an alternative mirror for a particular repository**. This overrides what's in the pom.xml.

For example, we can force Maven to use a single repository by mirroring all repository requests:

<**settings** xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/settings-1.0.0.xsd">

<**mirrors**>

<**mirror**>

<**id**>internal-baeldung-repository</**id**>

<**name**>Baeldung Internal Repo</**name**>

<**url**>https://baeldung.com/repo/maven2/</**url**>

<**mirrorOf**>\*</**mirrorOf**>

</**mirror**>

</**mirrors**>

</**settings**>

We may define only one mirror for a given repository and Maven will pick the first match. Normally, **we should use the official repository** distributed worldwide via CDN.

### 2.5. Servers

Defining repositories in the project pom.xml is a good practice. However, we shouldn't put security settings, such as credentials, into our source code repository with the pom.xml. Instead, we **define this** **secure information in the settings.xml file**:

<**settings** xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/settings-1.0.0.xsd">

<**servers**>

<**server**>

<**id**>internal-baeldung-repository</**id**>

<**username**>demo-user</**username**>

<**password**>dummy-password</**password**>

<**privateKey**>${user.home}/.ssh/bael\_key</**privateKey**>

<**passphrase**>dummy-passphrase</**passphrase**>

<**filePermissions**>664</**filePermissions**>

<**directoryPermissions**>775</**directoryPermissions**>

<**configuration**></**configuration**>

</**server**>

</**servers**>

</**settings**>

We should note that the ID of the server in the settings.xml needs to match the ID element of the repository mentioned in the pom.xml. The XML also allows us to use placeholders to pick up credentials from environment variables.

## 3. Profiles

The *profiles*element enables us to create multiple [*profile*](https://www.baeldung.com/maven-profiles) child elements differentiated by their ID child element. The profile element in the settings.xml is a truncated version of the same element available in the pom.xml.

It can contain only four child elements: activation, repositories, pluginRepositories, and properties. These elements configure the build system as a whole, instead of any particular project.

It's important to note that values from an active profile in settings.xml will **override any equivalent profile values in a pom.xml** or profiles.xml file. Profiles are matched by ID.

### 3.1. Activation

We can use profiles to modify certain values only under given circumstances. We can specify those circumstances using the activation element. Consequently, profile **activation occurs when all specified criteria are met**:

<**settings** xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/settings-1.0.0.xsd">

<**profiles**>

<**profile**>

<**id**>baeldung-test</**id**>

<**activation**>

<**activeByDefault**>false</**activeByDefault**>

<**jdk**>1.8</**jdk**>

<**os**>

<**name**>Windows 10</**name**>

<**family**>Windows</**family**>

<**arch**>amd64</**arch**>

<**version**>10.0</**version**>

</**os**>

<**property**>

<**name**>mavenVersion</**name**>

<**value**>3.0.7</**value**>

</**property**>

<**file**>

<**exists**>${basedir}/activation-file.properties</**exists**>

<**missing**>${basedir}/deactivation-file.properties</**missing**>

</**file**>

</**activation**>

</**profile**>

</**profiles**>

</**settings**>

There are four possible activators and not all of them need to be specified:

* jdk: activates based on the JDK version specified (ranges are supported)
* *os:*activates based on operating system properties
* *property:*activates the profile if Maven detects a specific property value
* *file:*activates the profile if a given filename exists or is missing

In order to check which profile will activate a certain build, we can use the Maven help plugin:

mvn help:active-profiles

The output will display currently active profiles for a given project:

[INFO] --- maven-help-plugin:3.2.0:active-profiles (default-cli) @ core-java-streams-3 ---

[INFO]

Active Profiles **for** Project 'com.baeldung.core-java-modules:core-java-streams-3:jar:0.1.0-SNAPSHOT':

The following profiles are active:

- baeldung-test (source: com.baeldung.core-java-modules:core-java-streams-3:0.1.0-SNAPSHOT)

### 3.2. Properties

Maven properties can be thought of as named placeholders for a certain value. The values are**accessible within a pom.xml file using the ${property\_name} notation**:

<**settings** xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/settings-1.0.0.xsd">

<**profiles**>

<**profile**>

<**id**>baeldung-test</**id**>

<**properties**>

<**user.project.folder**>${user.home}/baeldung-tutorials</**user.project.folder**>

</**properties**>

</**profile**>

</**profiles**>

</**settings**>

Four different types of properties are available in pom.xml files:

* Properties using the env prefix return an environment variable value, such as ${env.PATH}.
* Properties using the *project*prefix return a property value set in the project element of the pom.xml, such as ${project.version}.
* Properties using the *settings*prefix return the corresponding element’s value from the settings.xml, such as ${settings.localRepository}.
* We may reference all properties available via the System.getProperties method in Java directly, such as, ${java.home}.
* We may use properties set within a properties element without a prefix, such as ${junit.version}.

### 3.3. Repositories

Remote repositories contain collections of artifacts that Maven uses to populate our local repository. Different remote repositories may be needed for particular artifacts. Maven **searches the repositories enabled under the active profile**:

<**settings** xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/settings-1.0.0.xsd">

<**profiles**>

<**profile**>

<**id**>adobe-public</**id**>

<**repositories**>

<**repository**>

<**id**>adobe-public-releases</**id**>

<**name**>Adobe Public Repository</**name**>

<**url**>https://repo.adobe.com/nexus/content/groups/public</**url**>

<**releases**>

<**enabled**>true</**enabled**>

<**updatePolicy**>never</**updatePolicy**>

</**releases**>

<**snapshots**>

<**enabled**>false</**enabled**>

</**snapshots**>

</**repository**>

</**repositories**>

</**profile**>

</**profiles**>

</**settings**>

We can use the repository element to enable only release or snapshots versions of artifacts from a specific repository.

### 3.4. Plugin Repositories

There are two standard types of Maven artifacts, dependencies and plugins. As Maven plugins are a special type of artifact, we may **separate plugin repositories from the others**:

<**settings** xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/settings-1.0.0.xsd">

<**profiles**>

<**profile**>

<**id**>adobe-public</**id**>

<**pluginRepositories**>

<**pluginRepository**>

<**id**>adobe-public-releases</**id**>

<**name**>Adobe Public Repository</**name**>

<**url**>https://repo.adobe.com/nexus/content/groups/public</**url**>

<**releases**>

<**enabled**>true</**enabled**>

<**updatePolicy**>never</**updatePolicy**>

</**releases**>

<**snapshots**>

<**enabled**>false</**enabled**>

</**snapshots**>

</**pluginRepository**>

</**pluginRepositories**>

</**profile**>

</**profiles**>

</**settings**>

Notably, the structure of the pluginRepositories element is very similar to the repositories element.

### 3.5. Active Profiles

The activeProfiles element contains child elements that refer to a specific profile ID. **Maven automatically activates any profile referenced here**:

<**settings** xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/settings-1.0.0.xsd">

<**activeProfiles**>

<**activeProfile**>baeldung-test</**activeProfile**>

<**activeProfile**>adobe-public</**activeProfile**>

</**activeProfiles**>

</**settings**>

In this example, every invocation of mvn is run as though we've added -P baeldung-test,adobe-public to the command line.

## 4. Settings Level

A settings.xml file is usually found in a couple of places:

* Global settings in Mavens home directory: ${maven.home}/conf/settings.xml
* User settings in the user’s home: ${user.home}/.m2/settings.xml

If both files exist, their contents are merged. **Configurations from the user settings take precedence**.

### 4.1. Determine File Location

In order to determine the location of global and user settings, we can run Maven using the debug flag and search for “settings” in the output:

$ mvn -X clean | grep "settings"

[DEBUG] Reading global settings from C:\Program Files (x86)\Apache\apache-maven-3.6.3\bin\..\conf\settings.xml

[DEBUG] Reading user settings from C:\Users\Daniel Strmecki\.m2\settings.xml

### 4.2. Determine Effective Settings

We can use the Maven help plugin to **find out the contents of the combined global and user settings**:

mvn help:effective-settings

This describes the settings in XML format:

<**settings** xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/xsd/settings-1.0.0.xsd">

<**localRepository**>C:\Users\Daniel Strmecki\.m2\repository</**localRepository**>

<**pluginGroups**>

<**pluginGroup**>org.apache.tomcat.maven</**pluginGroup**>

<**pluginGroup**>org.apache.maven.plugins</**pluginGroup**>

<**pluginGroup**>org.codehaus.mojo</**pluginGroup**>

</**pluginGroups**>

</**settings**>

### 4.3. Override the Default Location

Maven also allows us to override the location of the global and user settings via the command line:

$ mvn clean --settings c:\user\user-settings.xml --global-settings c:\user\global-settings.xml

We can also use the shorter –s version of the same command:

$ mvn clean --s c:\user\user-settings.xml --gs c:\user\global-settings.xml

# Maven - Distribution Management

### **Table of Contents**

* [Maven - Distribution Management](https://datacadamia.com/maven/distribution_management#maven_-_distribution_management)
  + [About](https://datacadamia.com/maven/distribution_management#about)
  + [Articles Related](https://datacadamia.com/maven/distribution_management#articles_related)
  + [Example](https://datacadamia.com/maven/distribution_management#example)
    - [Remote Repository Location](https://datacadamia.com/maven/distribution_management#remote_repository_location)
    - [Credentials](https://datacadamia.com/maven/distribution_management#credentials)
    - [Deploy](https://datacadamia.com/maven/distribution_management#deploy)
  + [Documentation / Reference](https://datacadamia.com/maven/distribution_management#documentationreference)

## About

The [distribution Management](https://maven.apache.org/pom.html#Distribution_Management) [pom.xml](https://datacadamia.com/maven/pom.xml) section is responsible to define:

* the [remote repositories](https://datacadamia.com/maven/remote)
* how to deploy the project's site and documentation.

This data are used in the [deploy phase](https://datacadamia.com/maven/deploy)

## Articles Related

* [Maven - (Deploy|Distribution) Phase](https://datacadamia.com/maven/deploy)
* [Maven](https://datacadamia.com/maven/start)

## Example

Three steps are needed:

* give the repository location
* set the credentials
* deploy

### **Remote Repository Location**

with [Repository Manager - Nexus](https://datacadamia.com/infra/nexus), in the [pom.xml](https://datacadamia.com/maven/pom.xml), we set the repository locations

* for a [release](https://datacadamia.com/maven/release)
* and [snapshot](https://datacadamia.com/maven/snapshot)

<distributionManagement>

<repository>

<id>nexus</id>

<name>Releases</name>

<url>http://localhost:8081/repository/maven-releases</url>

</repository>

<snapshotRepository>

<id>nexus</id>

<name>Snapshot</name>

<url>http://localhost:8081/repository/maven-snapshots</url>

</snapshotRepository>

</distributionManagement>

XML

[Download pom.xml](https://datacadamia.com/maven/pom.xml)

### **Credentials**

and in [settings.xml](https://datacadamia.com/maven/settings.xml) the credentials

<settings>

...

<servers>

<server>

<id>nexus</id>

<username>admin</username>

<password>admin123</password>

</server>

</servers>

...

</settings>

XML

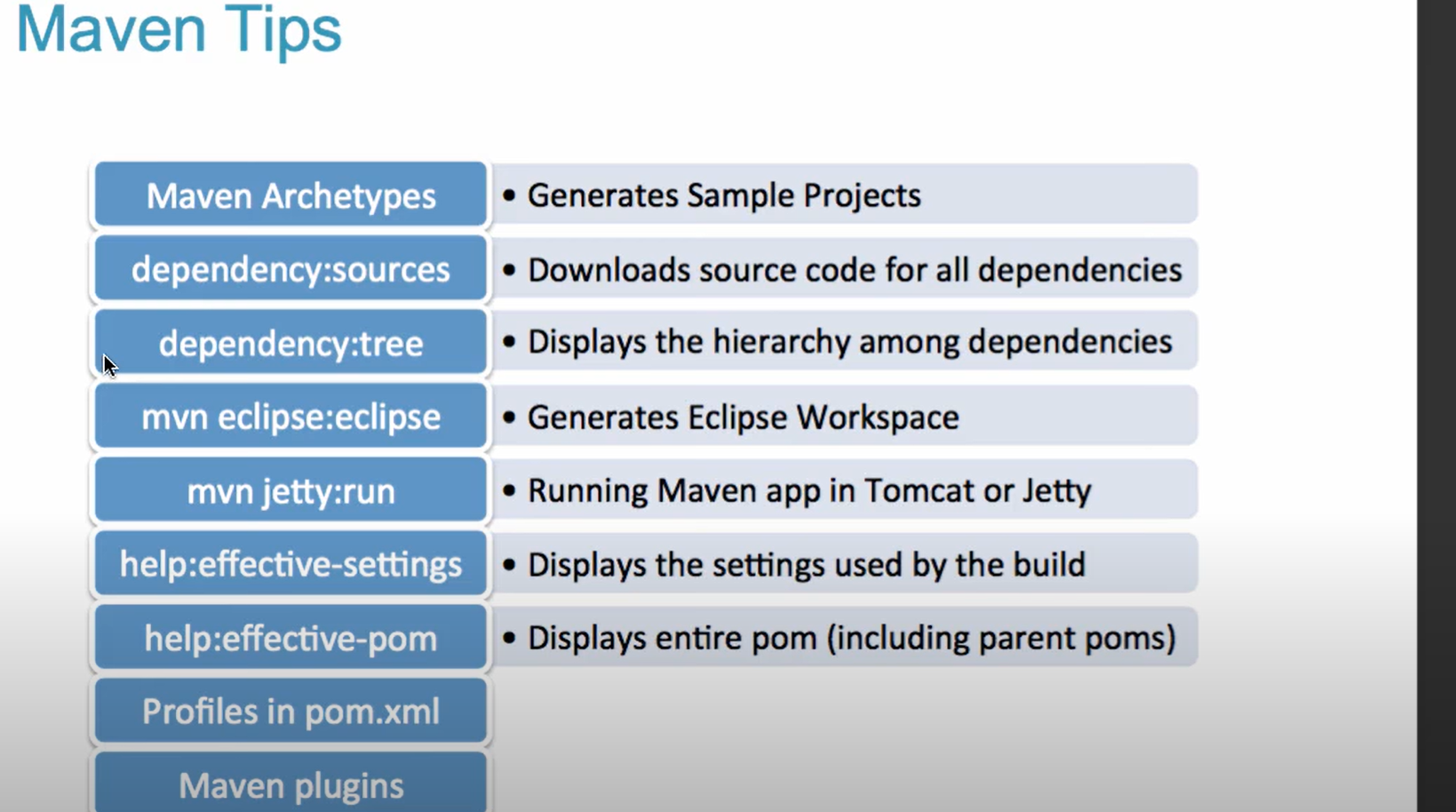
[Download settings.xml](https://datacadamia.com/maven/settings.xml)

See also: [Maven - (Password|Credentials)](https://datacadamia.com/maven/password)

### **Deploy**

[Maven - (Deploy|Distribution) Phase](https://datacadamia.com/maven/deploy)

mvn clean deploy



**How to setup maven:**

set JAVA\_HOME=C:\DC\software\OpenJDK-1.8\openjdk-8u282-b08

set PATH=%PATH%;%JAVA\_HOME%\bin

set M2\_HOME=C:\DC\software\apache-maven-3.6.1

set M2=%M2\_HOME%\bin

set PATH=%PATH%;%M2%;C:\Program Files\Git\bin;

set MAVEN\_OPTS=-Xms1024m -Xmx1024m

mvn clean install -T 3C -e -U -DskipTests

${group}:${artifact}:${type}:${version}

**for include or exclude**

mvn dependency:tree -Dincludes="\*:spring-web"

mvn dependency:tree -Dincludes=com.kcm.ssse:usg4j-ws

# Creating a Maven Repository

You’ll frequently want to share Liferay artifacts and modules with teammates or manage your repositories using a GUI. You can do this using [Sonatype Nexus](http://www.sonatype.org/nexus/). It’s a Maven repository management server that facilitates creating and managing release servers, snapshot servers, and proxy servers. There are several other Maven repository management servers you can use (for example, [Artifactory](https://www.jfrog.com/artifactory/)), but this tutorial focuses on Nexus.

To create a Maven repository using Nexus, download [Nexus](https://help.sonatype.com/display/NXRM2/Download) and follow the instructions on Nexus’ [Installation page](https://help.sonatype.com/display/NXRM2/Installing+and+Running) to install and start it.

To create your own repository using Nexus, follow these steps:

1. Open your web browser; navigate to your Nexus repository server (e.g., <http://localhost:8081/nexus>) and log in. The default user name is admin with password admin123.
2. Click on Repositories and navigate to Add… → Hosted Repository.

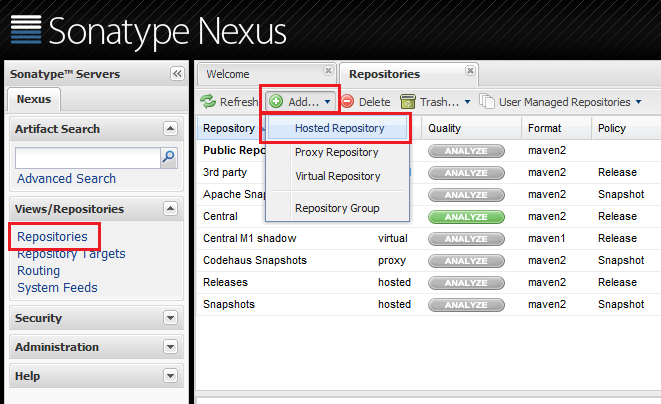


Figure 1: Adding a repository to hold your Liferay artifacts is easy with Nexus.

To learn more about each type of Nexus repository, read Sonatype’s [Managing Repositories](http://books.sonatype.com/nexus-book/reference/confignx-sect-manage-repo.html) guide.

1. Enter repository properties appropriate for the type of artifacts it will hold. If you’re installing release version artifacts into the repository, specify Release as the repository policy. Below are example repository property values:
   * **Repository ID:** liferay-releases
   * **Repository Name:** Liferay Release Repository
   * **Provider:** Maven2
   * **Repository Policy:** Release
2. Click Save.

You just created a Liferay Maven repository accessible from your Nexus repository server! Congratulations!

It’s also useful to create a Maven repository to hold snapshots of each Liferay project you create. Creating a snapshot repository is almost identical to creating a release repository. The only difference is that you specify Snapshot as its repository policy. For example, examine an example snapshot repository’s property values:

* **Repository ID:** liferay-snapshots
* **Repository Name:** Liferay Snapshot Repository
* **Provider:** Maven2
* **Repository Policy:** Snapshot

Voila! You’ve created a repository for your Liferay releases (i.e., liferay-releases) and Liferay snapshots (i.e., liferay-snapshots). To learn how to deploy your Liferay Maven artifacts to a Nexus repository, see the [Deploying Liferay Maven Artifacts to a Repository](https://help.liferay.com/hc/en-us/articles/360018164891-Deploying-Liferay-Maven-Artifacts-to-a-Repository-) tutorial.

Next, you’ll configure your new repository servers in your Maven settings to install artifacts to them.

## [Configuring Local Maven Settings](https://help.liferay.com/hc/en-us/articles/360018164871-Creating-a-Maven-Repository#configuring-local-maven-settings)

Before using your repository servers, you must specify them in your Maven environment settings. Your repository settings let Maven find the repository and retrieve and install artifacts. You can configure your local Maven settings in the [USER\_HOME]/.m2/settings.xml file.

You only need to configure a repository server if you’re sharing artifacts (e.g., Liferay artifacts and/or your modules) with others. If you’re automatically installing Liferay artifacts from the Central/Liferay Repository and aren’t interested in sharing artifacts, you don’t need a repository server specified in your Maven settings. You can find out more about installing artifacts from the Central Repository or Liferay’s own Nexus repository in the [Installing Liferay Maven Artifacts](https://help.liferay.com/hc/en-us/articles/360017885452-Installing-Liferay-Maven-Artifacts-) tutorial.

To configure your Maven environment to access your liferay-releases and liferay-snapshots repository servers, do the following:

1. Navigate to your [USER\_HOME]/.m2/settings.xml file. Create it if it doesn’t yet exist.
2. Provide settings for your repository servers. Here are contents from a settings.xml file that has liferay-releases and liferay-snapshots repository servers configured:
3. <?xml version="1.0"?>
4. <settings>
5. <servers>
6. <server>
7. <id>liferay-releases</id>
8. <username>admin</username>
9. <password>admin123</password>
10. </server>
11. <server>
12. <id>liferay-snapshots</id>
13. <username>admin</username>
14. <password>admin123</password>
15. </server>
16. </servers>
17. </settings>

The user name admin and password admin123 are the credentials of the default Nexus administrator account. If you changed these credentials for your Nexus server, make sure to update settings.xml with these changes.

# Deploying Liferay Maven Artifacts to a Repository

Deploying artifacts to a remote repository is important if you intend to share your Maven projects with others. First, you must have a remote repository that can hold deployed Maven artifacts. If you do not currently have a remote repository, see the [Creating a Maven Repository](https://help.liferay.com/hc/en-us/articles/360018164871-Creating-a-Maven-Repository-) tutorial to learn how you can set up a Nexus repository. Also make sure your [USER\_HOME]/.m2/settings.xml file specifies your remote repository’s ID, user name, and password.

To deploy to a remote repository, your Liferay project should be packaged using Maven. Maven provides a packaging command that creates an artifact (JAR) that can be easily deployed to your remote repository. You’ll learn how to do this with a Liferay portlet module.

Once you’ve created a deployable artifact, you’ll configure your module project to communicate with your remote repository and use Maven’s deploy command to send it on its way. Once your module project resides on the remote repository, other developers can configure your remote repository in their projects and set dependencies in their project POMs to reference it.

To follow this tutorial, you’ll need a Liferay module built with Maven. For demonstration purposes, this tutorial uses the portlet.ds sample module project. To follow along with this module, download the [portlet.ds](https://portal.liferay.dev/documents/113763090/114000186/portlet.ds.zip) Zip.

1. Create a folder anywhere on your machine to serve as the parent folder for your Liferay modules. Unzip the portlet.ds module project into that folder.
2. Create a pom.xml file inside this folder. Copy the following logic into the parent POM:
3. <?xml version="1.0" encoding="UTF-8"?>
4. <project
5. xmlns="http://maven.apache.org/POM/4.0.0"
6. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
8. >
9. <modelVersion>4.0.0</modelVersion>
10. <groupId>liferay.sample</groupId>
11. <artifactId>liferay.sample.maven</artifactId>
12. <version>1.0.0</version>
13. <name>Liferay Maven Module Projects</name>
14. <packaging>pom</packaging>
15. <distributionManagement>
16. <repository>
17. <id>liferay-releases</id>
18. <url>http://localhost:8081/nexus/content/repositories/liferay-releases</url>
19. </repository>
20. </distributionManagement>
21. <modules>
22. <module>portlet.ds</module>
23. </modules>
24. </project>

The tags <modelVersion> through <packaging> are POM tags that are used frequently in parent POMs. Visit Maven’s [POM Reference](https://maven.apache.org/pom.html) documentation for more information.

The <distributionManagement> tag specifies the deployment repository for all module projects residing in the parent folder. You should include the repository’s ID and URL. The above distributionManagement declaration is configured for the Liferay Nexus repository created in the [Creating a Maven Repository](https://help.liferay.com/hc/en-us/articles/360018164871-Creating-a-Maven-Repository-) tutorial. That tutorial also created the [USER\_HOME]/.m2/settings.xml, which specified the remote repository’s ID, user name, and password. Both the parent POM and settings.xml file’s repository declarations are required to deploy your modules to that remote repository.

Finally, you must list the modules residing in the parent folder that you want deployed using the <modules> tag. The portlet.ds module is specified within that tag.

1. Open the portlet.ds module’s pom.xml file. If you did not download the portlet.ds module project Zip, you can reference its POM below.
2. <project
3. xmlns="http://maven.apache.org/POM/4.0.0"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
6. >
7. <modelVersion>4.0.0</modelVersion>
8. <artifactId>portlet.ds</artifactId>
9. <version>1.0.0</version>
10. <packaging>jar</packaging>
11. <parent>
12. <groupId>liferay.sample</groupId>
13. <artifactId>liferay.sample.maven</artifactId>
14. <version>1.0.0</version>
15. <relativePath>../pom.xml</relativePath>
16. </parent>
17. <dependencies>
18. <dependency>
19. <groupId>javax.portlet</groupId>
20. <artifactId>portlet-api</artifactId>
21. <version>2.0</version>
22. <scope>provided</scope>
23. </dependency>
24. <dependency>
25. <groupId>org.osgi</groupId>
26. <artifactId>org.osgi.service.component.annotations</artifactId>
27. <version>1.3.0</version>
28. <scope>provided</scope>
29. </dependency>
30. </dependencies>
31. </project>

The portlet.ds module’s POM specifies its own attributes first, followed by the parent POM’s attributes. Declaring the <parent> tag like above links the portlet.ds module to its parent POM, which is necessary to deploy to the remote repository. Then the module’s dependencies are listed. These dependencies are downloaded from the Central Repository and installed to your local .m2 repository when you package the portlet.ds module.

1. Now that you’ve configured your parent POM and module POM, package your Maven project. Navigate to your module project (e.g., project.ds) using the command line and run the Maven package command:
2. mvn package

This downloads and installs all your module’s dependencies and packages the project into a JAR file. Navigate to your module project’s generated build folder (e.g., /target). You’ll notice there is a newly generated JAR file. This is the artifact you’ll deploy to your Nexus repository.

1. Run Maven’s deploy command to deploy your module project’s artifact to your configured remote repository.
2. mvn deploy

Your console shows output from the artifact being deployed into your repository server.

To verify that your artifact is deployed, navigate to the Repositories page of your Nexus server and select your repository. A window appears below showing the Liferay artifact now deployed to your repository.

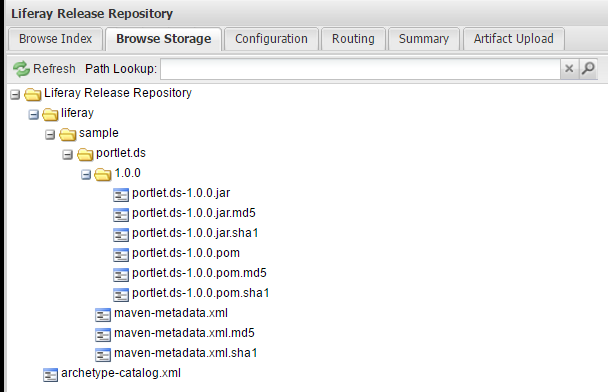


Figure 1: Your repository server now provides access to your Liferay Maven artifacts.