

Spring - Annotation Based Configuration

Advertisements

[Previous Page](http://www.tutorialspoint.com/spring/spring_beans_autowiring.htm)

[Next Page](http://www.tutorialspoint.com/spring/spring_java_based_configuration.htm)

Starting from Spring 2.5 it became possible to configure the dependency injection using **annotations**. So instead of using XML to describe a bean wiring, you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches.

Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file. So consider to have following configuration file in case you want to use any annotation in your Spring application.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- bean definitions go here -->

</beans>

Once <context:annotation-config/> is configured, you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors. Let us see few important annotations to understand how they work:

|  |
| --- |
| [**@Required**](http://www.tutorialspoint.com/spring/spring_required_annotation.htm)  The @Required annotation applies to bean property setter methods. |
| 2 | [**@Autowired**](http://www.tutorialspoint.com/spring/spring_autowired_annotation.htm)  The @Autowired annotation can apply to bean property setter methods, non-setter methods, constructor and properties. |
| 3 | [**@Qualifier**](http://www.tutorialspoint.com/spring/spring_qualifier_annotation.htm)  The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifiying which exact bean will be wired. |
| 4 | [**JSR-250 Annotations**](http://www.tutorialspoint.com/spring/spring_jsr250_annotations.htm)  Spring supports JSR-250 based annotations which include @Resource, @PostConstruct and @PreDestroy annotations. |

# dependency

http://img.businessdictionary.com/term_speaker.jpg

## Definition

[Relationship](http://www.businessdictionary.com/definition/relationship.html) between [conditions](http://www.businessdictionary.com/definition/condition.html), [events](http://www.businessdictionary.com/definition/events.html), or [tasks](http://www.businessdictionary.com/definition/task.html) such that one cannot begin or be-completed until one or more other conditions, events, or tasks have occurred, begun, or [completed](http://www.businessdictionary.com/definition/completed.html).

### What is a POM?

A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. It contains default values for most projects. Examples for this is the build directory, which is target; the source directory, which issrc/main/java; the test source directory, which is src/test/java; and so on.

The POM was renamed from project.xml in Maven 1 to pom.xml in Maven 2. Instead of having a maven.xml file that contains the goals that can be executed, the goals or plugins are now configured in the pom.xml. When executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, then executes the goal.

Some of the configuration that can be specified in the POM are the project dependencies, the plugins or goals that can be executed, the build profiles, and so on. Other information such as the project version, description, developers, mailing lists and such can also be specified.

## Introduction to Repositories

### Artifact Repositories

A repository in Maven is used to hold build artifacts and dependencies of varying types.

There are strictly only two types of repositories: local and remote. The local repository refers to a copy on your own installation that is a cache of the remote downloads, and also contains the temporary build artifacts that you have not yet released.

Remote repositories refer to any other type of repository, accessed by a variety of protocols such as file:// and http://. These repositories might be a truly remote repository set up by a third party to provide their artifacts for downloading (for example, [repo.maven.apache.org](http://repo.maven.apache.org/maven2/) and [uk.maven.org](http://uk.maven.org/maven2/) house Maven's central repository). Other "remote" repositories may be internal repositories set up on a file or HTTP server within your company, used to share private artifacts between development teams and for releases.

The local and remote repositories are structured the same way so that scripts can easily be run on either side, or they can be synced for offline used. In general use, the layout of the repositories is completely transparent to the Maven user, however.

### Why not Store JARs in CVS?

It is not recommended that you store your JARs in CVS. Maven tries to promote the notion of a user local repository where JARs, or any project artifacts, can be stored and used for any number of builds. Many projects have dependencies such as XML parsers and standard utilities that are often replicated in typical builds. With Maven these standard utilities can be stored in your local repository and shared by any number of builds.

This has the following advantages:

* **It uses less storage** - while a repository is typically quite large, because each JAR is only kept in the one place it is actually saving space, even though it may not seem that way
* **It makes checking out a project quicker** - initial checkout, and to a small degree updating, a project will be faster if there are no large binary files in CVS. While they may need to be downloaded again afterwards anyway, this only happens once and may not be necessary for some common JARs already in place.
* **No need for versioning** - CVS and other source control systems are designed for versioning files, but external dependencies typically don't change, or if they do their filename changes anyway to indicate the new version. Storing these in CVS doesn't have any added benefit over keeping them in a local artifact cache.

### Using Repositories

In general, you should not need to do anything with the local repository on a regular basis, except clean it out if you are short on disk space (or erase it completely if you are willing to download everything again).

For the remote repositories, they are used for both downloading and uploading (if you have the permission to do so).

#### Downloading from a Remote Repository

Downloading in Maven is triggered by a project declaring a dependency that is not present in the local repository (or for a SNAPSHOT, when the remote repository contains one that is newer). By default, Maven will download from the [central](http://repo.maven.apache.org/maven2/) repository.

To override this, you need to specify a mirror as shown in [Using Mirrors for Repositories](https://maven.apache.org/guides/mini/guide-mirror-settings.html)

You can set this in your settings.xml file to globally use a certain mirror, however note that it is common for a project to customise the repository in their pom.xml and that your setting will take precedence. If you find that dependencies are not being found, check you have not overridden the remote repository.

For more information on dependencies, see [Dependency Mechanism](https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html).

#### Using Mirrors for the Central Repository

There are several official Central repositories geographically distributed. You can make changes to your settings.xml file to use one or more mirrors. Instructions for this can be found in the guide [Using Mirrors for Repositories](https://maven.apache.org/guides/mini/guide-mirror-settings.html).

### Building Offline

If you are temporarily disconnected from the internet and you need to build your projects offline you can use the offline switch on the CLI:

1. mvn -o package

Note that many plugins will honour the offline setting and not perform any operations that would connect to the internet. Some examples are resolving Javadoc links and link checking the site.

### Uploading to a Remote Repository

While this is possible for any type of remote repository, you must have the permission to do so. To have someone upload to the central Maven repository, see [Repository Center](https://maven.apache.org/repository/index.html).

## Internal Repositories

When using Maven, particularly in a corporate environment, connecting to the internet to download dependencies is not acceptable for security, speed or bandwidth reasons. For that reason, it is desirable to set up an internal repository to house a copy of artifacts, and to publish private artifacts to.

Such an internal repository can be downloaded from using HTTP or the file system (using a file:// URL), and uploaded to using SCP, FTP, or a file copy.

Note that as far as Maven is concerned, there is nothing special about this repository: it is another **remote repository** that contains artifacts to download to a user's local cache, and is a publish destination for artifact releases.

Additionally, you may want to share the repository server with your generated project sites. For more information on creating and deploying sites, see [Creating a Site](https://maven.apache.org/guides/mini/guide-site.html).

### Setting up the Internal Repository

To set up an internal repository just requires that you have a place to put it, and then start copying required artifacts there using the same layout as in a remote repository such as [repo.maven.apache.org](http://repo.maven.apache.org/maven2/).

It is *not* recommended that you scrape or rsync:// a full copy of central as there is a large amount of data there and doing so will get you banned. You can use a program such as those described on the [Repository Management](https://maven.apache.org/repository-management.html) page to run your internal repository's server, to download from the internet as required and then hold the artifacts in your internal repository for faster downloading later.

The other options available are to manually download and vet releases, then copy them to the internal repository, or to have Maven download them for a user, and manually upload the vetted artifacts to the internal repository which is used for releases. This step is the only one available for artifacts where the license forbids their distribution automatically, such as several J2EE JARs provided by Sun. Refer to the [Guide to coping with SUN JARs](https://maven.apache.org/guides/mini/guide-coping-with-sun-jars.html) document for more information.

It should be noted that Maven intends to include enhanced support for such features in the future, including click through licenses on downloading, and verification of signatures.

### Using the Internal Repository

Using the internal repository is quite simple. Simply make a change to add a repositories element:

1. <project>
2. ...
3. <repositories>
4. <repository>
5. <id>my-internal-site</id>
6. <url>http://myserver/repo</url>
7. </repository>
8. </repositories>
9. ...
10. </project>

If your internal repository requires authentication, the id element can be used in your [settings](https://maven.apache.org/settings.html#Servers) file to specify login information.

### Deploying to the Internal Repository

One of the most important reasons to have one or more internal repositories is to be able to publish your own private releases to share.

To publish to the repository, you will need to have access via one of SCP, SFTP, FTP, WebDAV, or the filesystem. Connectivity is accomplished with the various [wagons](http://maven.apache.org/wagon/wagon-providers/index.html). Some wagons may need to be added as [extension](http://maven.apache.org/ref/current/maven-model/maven.html#class_extension) to your build.

# Where is Maven local repository?

By [mkyong](https://www.mkyong.com/author/mkyong/) | January 23, 2009 | Updated : December 13, 2012 | Viewed : 357,145 times +1,806 pv/w

The maven local repository is a local folder that is used to store all your project’s dependencies (plugin jars and other files which are downloaded by Maven). In simple, when you build a Maven project, all dependency files will be stored in your Maven local repository.

By default, Maven local repository is default to .m2 folder :

1. Unix/Mac OS X – ~/.m2
2. Windows – C:\Documents and Settings\{your-username}\.m2

## 1. Update Maven Local Repository

Normally, I will change the default local repository folder from default .m2 to another more meaningful name, for example, maven-repo.

Find **{M2\_HOME}\conf\setting.xml**, update localRepository to something else.

{M2\_HOME}\conf\setting.xml

<settings>

<!-- localRepository

| The path to the local repository maven will use to store artifacts.

|

| Default: ~/.m2/repository

<localRepository>/path/to/local/repo</localRepository>

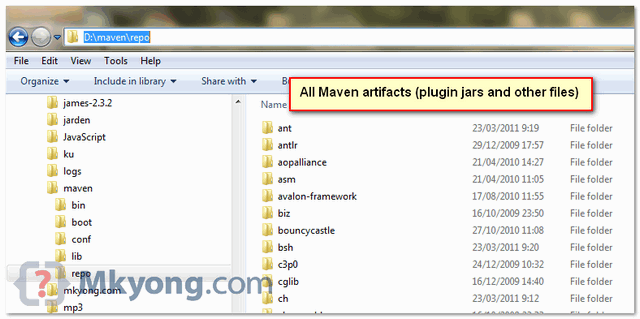
-->

<localRepository>D:/maven\_repo</localRepository>

## 2. Saved it

Done, your new Maven local repository is now changed to D:/maven\_repo.

See figure :



## Guide to naming conventions on groupId, artifactId and version

* **groupId** will identify your project uniquely across all projects, so we need to enforce a naming schema. It has to follow the package name rules, what means that has to be at least as a domain name you control, and you can create as many subgroups as you want. Look at [More information about package names](http://java.sun.com/docs/books/jls/third_edition/html/packages.html#7.7).

eg. org.apache.maven, org.apache.commons

A good way to determine the granularity of the groupId is to use the project structure. That is, if the current project is a multiple module project, it should append a new identifier to the parent's groupId.

eg. org.apache.maven, org.apache.maven.plugins, org.apache.maven.reporting

* **artifactId** is the name of the jar without version. If you created it then you can choose whatever name you want with lowercase letters and no strange symbols. If it's a third party jar you have to take the name of the jar as it's distributed.

eg. maven, commons-math

* **version** if you distribute it then you can choose any typical version with numbers and dots (1.0, 1.1, 1.0.1, ...). Don't use dates as they are usually associated with SNAPSHOT (nightly) builds. If it's a third party artifact, you have to use their version number whatever it is, and as strange as it can look.

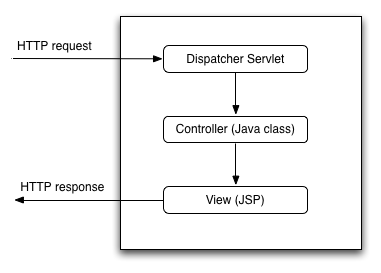
eg. 2.0, 2.0.1, 1.3.1

# [Why do we use web.xml?](http://stackoverflow.com/questions/4538846/why-do-we-use-web-xml)

|  |  |
| --- | --- |
| 35down voteaccepted | Generally speaking, this is the configuration file of web applications in java. It instructs the servlet container (tomcat for ex.) which classes to load, what parameters to set in the context, and how to intercept requests coming from browsers.  There you specify:   * what servlets (and filters) you want to use and what URLs you want to map them to * listeners - classes that are notified when some events happen (context starts, session created, etc) * configuration parameters (context-params) * error pages, welcome files * security constriants   In servlet 3.0 many of the web.xml parts are optional. These configurations can be done via annotations (@WebServlet, @WebListener) |

# [What is Dispatcher Servlet in Spring?](http://stackoverflow.com/questions/2769467/what-is-dispatcher-servlet-in-spring)

n this image (which I got from [**here**](http://maestric.com/wiki/lib/exe/fetch.php?w=&h=&cache=cache&media=java:spring:spring_mvc.png)), HTTP request sends something to Dispatcher Servlet.



|  |  |
| --- | --- |
| 92down voteaccepted | The job of the *DispatcherServlet* is to take an incoming URI and find the right combination of handlers (generally methods on *Controller* classes) and views (generally JSPs) that combine to form the page or resource that's supposed to be found at that location.  I might have   * a file /WEB-INF/jsp/pages/Home.jsp * and a *method* on a class * @RequestMapping(value="/pages/Home.html") * private ModelMap buildHome() { * return somestuff;   }  The *Dispatcher servlet* is the bit that "knows" to call that method when a browser requests the page, and to combine its results with the matching JSP file to make an html document.  How it accomplishes this varies widely with configuration and Spring version.  There's also no reason the end result has to be web pages. It can do the same thing to locate *RMI*end points, handle *SOAP* requests, anything that can come into a servlet. |

# [What is WEB-INF used for in a Java EE web application?](http://stackoverflow.com/questions/19786142/what-is-web-inf-used-for-in-a-java-ee-web-application)

|  |  |
| --- | --- |
| up vote105down voteaccepted | The [Servlet 2.4 specification](http://download.oracle.com/otn-pub/jcp/servlet-2.4-fr-spec-oth-JSpec/servlet-2_4-fr-spec.pdf) says this about WEB-INF (page 70):  A special directory exists within the application hierarchy named WEB-INF. This directory contains all things related to the application that aren’t in the document root of the application. **The WEB-INFnode is not part of the public document tree of the application**. No file contained in the WEB-INF directory may be served directly to a client by the container. However, the contents of theWEB-INF directory are visible to servlet code using the getResource and getResourceAsStreammethod calls on the ServletContext, and may be exposed using the RequestDispatcher calls.  This means that WEB-INF resources are accessible to the resource loader of your Web-Application and not directly visible for the public.  This is why a lot of projects put their resources like JSP files, JARs/libraries and their own class files or property files or any other sensitive information in the WEB-INF folder. Otherwise they would be accessible by using a simple static URL (usefull to load CSS or Javascript for instance).  Your JSP files can be anywhere though from a technical perspective. For instance in Spring you can configure them to be in WEB-INF explicitly:  <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"  p:prefix="/WEB-INF/jsp/"  p:suffix=".jsp" >  </bean>  As to the WEB-INF/classes and WEB-INF/lib folders, you've there a Maven project structure. Those folders are by default not visible in a Maven project as it will automatically create and fill those folders during the build of the WAR file based on the source files and the pom dependencies. They are only visible in the produced WAR file. In a "standard" Eclipse dynamic web project, the WEB-INF/lib folder is however visible and one without Maven would need to manually drop physical JARs in there in order to install dependencies. |

# [How do I add a resources folder to my Java project in eclipse](http://stackoverflow.com/questions/27934796/how-do-i-add-a-resources-folder-to-my-java-project-in-eclipse)

|  |  |
| --- | --- |
| 11down vote[favorite](http://stackoverflow.com/questions/27934796/how-do-i-add-a-resources-folder-to-my-java-project-in-eclipse)  2 | I want to have a place to store my image files to use in my java project (a really simple class that just loads an image onto a panel). I have looked everywhere and cannot find how to do this. How do I do this? I have tried adding a new folder to the project, adding a new class folder to the project, and adding a new source folder to the project. No matter what I do, I always get a IOException. The folders always say they are on the build path, so I'm not sure what to do. |

# [Adding folder in eclipse in src directory without making it package](http://stackoverflow.com/questions/8025371/adding-folder-in-eclipse-in-src-directory-without-making-it-package)

Eclipse forces you distinguish between source directories and ordinary folders. Any subdirectories in a source folder will be considered a package.

In your case, you can create an ordinary folder outside of src/ to prevent the subdirectories from being interpreted as packages.

Alternatively, you can modify the project properties to have src/ be considered an ordinary directory, and put a source directory within it.

You can manage which directories in a project are considered source directories by:

1. Right-clicking your project, then click Properties.
2. In the left pane, click Java Build Path. In the right pane, select the Source tab.
3. Here you can add/edit/remove source folders.

# [Eclipse not deploying my web app properly](http://stackoverflow.com/questions/18226414/eclipse-not-deploying-my-web-app-properly)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| up vote7down voteaccepted | Since it is a maven project .  you might have converted maven project to dynamic web project I guess.  Follow the steps  1)Go to project properties (right click at last click peoperties)  2)click/select Deployment assembly at the left side  3)the root(/) deploy path may be pointing to src/main/webapp , select this and click remove button  4)click on add button , select a directive type folder, select your WebContent folder  Now try to clean and run your application on server.  **Solution 2**  1) Actually you need to copy/put all your **WebContent** *files and folder* to src/main/webapp  and restart your server.  Note that the maven dynamic web project it always look in to src/main/webapp by default.  Hope it helps Introduction to the Standard Directory Layout Having a common directory layout would allow for users familiar with one Maven project to immediately feel at home in another Maven project. The advantages are analogous to adopting a site-wide look-and-feel.  The next section documents the directory layout expected by Maven and the directory layout created by Maven. Please try to conform to this structure as much as possible; however, if you can't these settings can be overridden via the project descriptor.   |  |  | | --- | --- | | src/main/java | Application/Library sources | | src/main/resources | Application/Library resources | | src/main/resources-filtered | Application/Library resources which are filtered. (Starting with Maven 3.4.0, not yet released.) | | src/main/filters | Resource filter files | | src/main/webapp | Web application sources | | src/test/java | Test sources | | src/test/resources | Test resources | | src/test/resources-filtered | Test resources which are filtered by default. (Starting with Maven 3.4.0, not yet released.) | | src/test/filters | Test resource filter files | | src/it | Integration Tests (primarily for plugins) | | src/assembly | Assembly descriptors | | src/site | Site | | LICENSE.txt | Project's license | | NOTICE.txt | Notices and attributions required by libraries that the project depends on | | README.txt | Project's readme |   At the top level files descriptive of the project: a pom.xml file In addition, there are textual documents meant for the user to be able to read immediately on receiving the source: README.txt, LICENSE.txt, etc.  There are just two subdirectories of this structure: src and target. The only other directories that would be expected here are metadata like CVS, .git or .svn, and any subprojects in a multiproject build (each of which would be laid out as above).  The target directory is used to house all output of the build.  The src directory contains all of the source material for building the project, its site and so on. It contains a subdirectory for each type: main for the main build artifact,test for the unit test code and resources, site and so on.  Within artifact producing source directories (ie. main and test), there is one directory for the language java (under which the normal package hierarchy exists), and one for resources (the structure which is copied to the target classpath given the default resource definition).  If there are other contributing sources to the artifact build, they would be under other subdirectories: for example src/main/antlr would contain Antlr grammar definition files. |