# Maven's Objectives

Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. In order to attain this goal there are several areas of concern that Maven attempts to deal with:

* Making the build process easy
* Providing a uniform build system
* Providing quality project information
* Providing guidelines for best practices development
* Allowing transparent migration to new features

# What is Maven Not?

You may have heard some of the following things about Maven:

* Maven is a site and documentation tool
* Maven extends Ant to let you download dependencies
* Maven is a set of reusable Ant scriptlets

# Feature Summary

The following are the key features of Maven in a nutshell:

* Simple project setup that follows best practices - get a new project or module started in seconds
* Consistent usage across all projects means no ramp up time for new developers coming onto a project
* Superior dependency management including automatic updating, dependency closures (also known as transitive dependencies)
* Able to easily work with multiple projects at the same time
* A large and growing repository of libraries and metadata to use out of the box, and arrangements in place with the largest Open Source projects for real-time availability of their latest releases
* Extensible, with the ability to easily write plugins in Java or scripting languages
* Instant access to new features with little or no extra configuration
* Ant tasks for dependency management and deployment outside of Maven
* Model based builds: Maven is able to build any number of projects into predefined output types such as a JAR, WAR, or distribution based on metadata about the project, without the need to do any scripting in most cases.
* Coherent site of project information: Using the same metadata as for the build process, Maven is able to generate a web site or PDF including any documentation you care to add, and adds to that standard reports about the state of development of the project. Examples of this information can be seen at the bottom of the left-hand navigation of this site under the "Project Information" and "Project Reports" submenus.
* Release management and distribution publication: Without much additional configuration, Maven will integrate with your source control system such as CVS and manage the release of a project based on a certain tag. It can also publish this to a distribution location for use by other projects. Maven is able to publish individual outputs such as a JAR, an archive including other dependencies and documentation, or as a source distribution.
* Dependency management: Maven encourages the use of a central repository of JARs and other dependencies. Maven comes with a mechanism that your project's clients can use to download any JARs required for building your project from a central JAR repository much like Perl's CPAN. This allows users of Maven to reuse JARs across projects and encourages communication between projects to ensure that backward compatibility issues are dealt with.

# Creating a Maven Project

mvn archetype:create -DgroupId=com.mycompany.app -DartifactId=my-app

We just executed the Maven goal archetype:create, and passed in various parameters to that goal. The prefix archetype is the [plugin](http://maven.apache.org/plugins/index.html) that contains the goal. If you are familiar with [Ant](http://ant.apache.org), you may concieve of this as similar to a task. This goal created a simple project based upon an archetype. Suffice it to say for now that a plugin is a collection of goals with a general common purpose. For example the jboss-maven-plugin, whose purpose is "deal with various jboss items".

# Build

mvn package

Rather than a goal, this is a *phase*. A phase is a step in the build lifecycle, which is an ordered sequence of phases. When a phase is given, Maven will execute every phase in the sequence up to and including the one defined. For example, if we execute the *compile* phase, the phases that actually get executed are:

1. validate
2. generate-sources
3. process-sources
4. generate-resources
5. process-resources
6. compile

# Maven Phases

Although hardly a comprehensive list, these are the most common *default* lifecycle phases executed.

* **validate**: validate the project is correct and all necessary information is available
* **compile**: compile the source code of the project
* **test**: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
* **package**: take the compiled code and package it in its distributable format, such as a JAR.
* **integration-test**: process and deploy the package if necessary into an environment where integration tests can be run
* **verify**: run any checks to verify the package is valid and meets quality criteria
* **install**: install the package into the local repository, for use as a dependency in other projects locally
* **deploy**: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

There are two other Maven lifecycles of note beyond the *default* list above. They are

* **clean**: cleans up artifacts created by prior builds
* **site**: generates site documentation for this project

Phases are actually mapped to underlying goals. The specific goals executed per phase are dependent upon the packaging type of the project. For example, *package* executes *jar:jar* if the project type is a JAR, and *war:war* is the project type is - you guessed it - a WAR.

# Maven Archetype

In short, Archetype is a Maven project templating toolkit. An archetype is defined as *an original pattern or model from which all other things of the same kind are made*.

# Creating an Archetype

Creating an archetype is a pretty straight forward process. An archetype is a very simple plugin that contains the project prototype you wish to create. An archetype is made up of:

* An archetype descriptor (archetype.xml in directory: src/main/resources/META-INF/). It lists all the files that will be contained in the archetype and categorizes them so they can be processed correctly by the archetype generation mechanism.
* the prototype files that are copied by the archetype (directory: src/main/resources/archetype-resources/)
* the prototype pom (pom.xml in: src/main/resources/archetype-resources)
* A pom for the archetype (pom.xml in the archetype's root directory).

# Introduction to the 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 are the build directory, which is target; the source directory, which is src/main/java; the test source directory, which is src/main/test; and so on.

### Super POM

The Super POM is Maven's default POM. All POMs extend the Super POM unless explicitly set, meaning the configuration specified in the Super POM is inherited by the POMs you created for your projects.

A POM requires that its groupId, artifactId, and version be configured. These three values form the project's fully qualified artifact name. This is in the form of <groupId>:<artifactId>:<version>. As for the example above, its fully qualified artifact name is "com.mycompany.app:my-app:1".

If the configuration details are not specified, Maven will use their defaults. One of these default values is the packaging type. Every Maven project has a packaging type. If it is not specified in the POM, then the default value "jar" would be used.

Furthermore, as you can see that in the minimal POM, the repositories were not specified. If you build your project using the minimal POM, it would inherit the repositories configuration in the Super POM. Therefore when Maven sees the dependencies in the minimal POM, it would know that these dependencies will be downloaded from http://repo1.maven.org/maven2 which was specified in the Super POM.