maven

Maven v.1.1

Project Documentation

Table of Contents

1 About Maven 1.x
1.1 Welcome . . . 1
1.2 What is Maven? . . . 4
1.2.1 Feature Summary . . . 7
1.3 Download . . . 8
1.3.1 Release Notes . . . 9
1.3.2 Compatibility . . . 10
1.3.3 Road Map . . . 14

2 User's Guide
2.1 Getting Started . . . 15
2.1.1 Installing . . . 16
2.1.2 Building a Project . . . 18
2.1.3 Creating a Project . . . 22
2.1.4 Adapting a Project . . . 27
2.1.5 Migrating from Ant . . . 28
2.2 Using Maven . . . 33
2.2.1 Building JARs . . . 34
2.2.2 Resources . . . 35
2.2.3 Unit Testing . . . 37
2.2.4 Creating a Site . . . 38
2.2.5 Source Control . . . 42
2.2.6 Making Releases . . . 44
2.2.7 Web Applications . . . 47
2.2.8 Multiple Modules . . . 50
2.2.9 Handling Dependencies . . . 53
2.2.10 Best Practices . . . 56
2.2.11 IDE Integration . . . 60
2.2.11.1 IntelliJ IDEA . . . 61
2.3 Customising Maven . . . 62
2.3.1 Scripting . . . 65

2.3.2 Writing a Plugin . . . 70
2.3.3 Sharing Plugins . . . 76
2.4 Repositories . . . 78
2.4.1 Internal Repositories . . . 81
2.4.2 Sun JAR Names . . . 83
2.4.3 Uploading to Ibiblio . . . 85
2.5 Reference . . . 88
2.5.1 Glossary . . . 89
2.5.2 Conventions . . . 90
2.5.3 Project Descriptor . . . 92
2.5.4 Properties Reference . . . 93
2.5.5 Command Line . . . 97
2.6 Plugins . . . 99
2.6.1 History . . . 100
2.6.2 Other 3rd Party Plugins . . . 105
2.7 FAQ . . . 107

3 **Resources**

3.1 Powered by Maven . . . 121
3.2 Related links . . . 123
3.2.1 Books and Articles . . . 124
3.3 Maven projects . . . 126
3.4 Related projects . . . 127

1.1 Welcome

Welcome to Maven

Latest News

Maven 1.1 released!

25th June 2007

- Download
- Installation Instructions
- Release Notes
- Getting Started

New plugin releases

23th May 2007

- test plugin 1.8.2
- artifact plugin 1.9.1

Maven 1.1-RC1 released!

11th May 2007

- Download
- Installation Instructions
- Release Notes
- Getting Started

New plugin releases

8th May 2007

- artifact plugin 1.9
- changelog plugin 1.9.2
- dist plugin 1.7.1
- eclipse plugin 1.12
- ejb plugin 1.7.3
- linkcheck plugin 1.4.1
- multiproject plugin 1.5.1
- nsis plugin 2.1
- test plugin 1.8.1
- war plugin 1.6.3
- xdoc plugin 1.10.1

All news ...

Known issues

- Still using Maven 1.0.x and find that dependencies cannot be downloaded?. Check this news entry first.

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.

These pages are all about Maven 1.x. The latest version of Maven is the 2.0 tree, which is a complete rewrite of the original Maven application. For more information on Maven 2, see the main Maven site.

Note that **no further development** is planned for the Maven-1.0 branch, and Maven-1.1 is in maintenance mode, i.e. development is restricted to support and bug fixes.

Getting Started with Maven

Need Maven to build a project you have downloaded?

Read our Quick Start guide on how to get and use Maven for a project that is already set up for Maven.

Want to get Maven up and running on a new project?

Give it the Ten Minute Test! This will show you how to set up a simple project that uses Maven.

Want to adapt an existing project to use Maven?

Read our Adapting a Project guide for a guide on how to introduce Maven to an existing build.

Familiar with Ant, and want to see how Maven relates to it?

Read our Maven for Ant Users explanatory document.

Help and Feedback

If you have any questions, suggestions or comments about Maven or this website, please feel free to post them to the Maven users mailing list.

Release Information

Stable

The current stable release for maven 1.x is version 1.1, obtainable from the download page.

Development/Head

Obtain the latest code from Subversion. For more information, see Building from Source and compatibility information.

For documentation, Continuum regularly generates "maven site" for the head revisions and publishes them to the Continuum staging sites.

1.2 What is Maven?

Introduction

Maven was originally started as an attempt to simplify the build processes in the Jakarta Turbine project. There were several projects each with their own Ant build files that were all slightly different and JARs were checked into CVS. We wanted a standard way to build the projects, a clear definition of what the project consisted of, an easy way to publish project information and a way to share JARs across several projects.

What resulted is a tool that can now be used for building and managing any Java-based project. We hope that we have created something that will make the day-to-day work of Java developers easier and generally help with the comprehension of any Java-based project.

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

Making the build process easy

While using Maven doesn't eliminate the need to know about the underlying mechanisms, Maven does provide a lot of shielding from the details.

Providing a uniform build system

Maven allows a project to build using its project object model (POM) and a set of plugins that are shared by all projects using Maven, providing a uniform build system. Once you familiarize yourself with how one Maven project builds you automatically know how all Maven projects build saving you immense amounts of time when trying to navigate many projects.

Providing quality project information

Maven provides plenty of useful project information that is in part taken from your POM and in part generated from your project's sources. For example, Maven can provide:

- Change log document created directly from source control.
- Cross referenced sources

- Mailing lists
- Dependency list
- Unit test reports including coverage

As Maven improves the information set provided will improve, all of which will be transparent to users of Maven.

Other products can also provide Maven plugins to allow their set of project information alongside some of the standard information given by Maven, all still based from the POM.

Providing guidelines for best practices development

Maven aims to gather current principles for best practices development, and make it easy to guide a project in that direction.

For example, specification, execution, and reporting of unit tests are part of the normal build cycle using Maven. Current unit testing best practices were used as guidelines:

- Keeping your test source code in a separate, but parallel source tree
- Using test case naming conventions to locate and execute tests
- Have test cases setup their environment and don't rely on customizing the build for test preparation.

Maven also aims to assist in project workflow such as release management and issue tracking.

Maven also suggests some guidelines on how to layout your project's directory structure so that once you learn the layout you can easily navigate any other project that uses Maven and the same defaults.

Allowing transparent migration to new features

Maven provides an easy way for Maven clients to update their installations so that they can take advantage of any changes that been made to Maven itself.

Installation of new or updated plugins from third parties or Maven itself has been made trivial for this reason.

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

While Maven does these things, as you can read above in the "What is Maven?" section, these are not the only features Maven has, and it's objectives are quite different.

Maven does encourage best practices, but we realise that some projects may not fit with these ideals for historical reasons. While Maven is designed to be flexible, to an extent, in these situations and to the needs of different projects, it can not cater to every situation without making compromises to the

integrity of its objectives.

If you decide to use Maven, and have an unusual build structure that you cannot reorganise, you may have to forgo some features or the use of Maven altogether.

How Does Maven Compare to Ant?

More information on this topic can be found in Maven for Ant Users.

1.2.1 Feature Summary

Feature Summary

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

Feature	Description
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 any 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 backward compatibility issues are dealt with.
Gump integration	Integration with Gump. For those who are not familiar with Gump it is a tool used at Apache to help projects maintain backward compatibility with their clients. If you have a Maven project descriptor then you can easily participate in nightly Gump builds that will help your project stay abreast of the impact your changes actually have in Java developer community. We are working on our own massive build tool but integration with Gump comes at no cost to Maven users.

Curious who else is using Maven? A few of the many projects doing so can be seen at the Powered by Maven page.

For more information on Maven's objectives, see What is Maven?.

If you currently use Ant, you might be interested to read about Maven from an Ant perspective.

1.3 Download

Download Maven 1.x

Maven is currently distributed in several formats for your convenience. For instructions on how to install maven, see the Installation Instructions. You can find both these and older releases here.

Latest Stable Release: Download Maven 1.1

Please read the Release Notes. In particular, note that you **must adjust your path** when installing side by side with Maven 1.0.2.

- Windows Installer (checksum) (PGP)
- .tar.bz2 archive (checksum) (PGP)
- .tar.gz archive (checksum) (PGP)
- .zip archive (checksum) (PGP)

Maven is distributed under the Apache License, version 2.0.

Previous Releases

All previous releases of Maven can be found in the archives.

Plugins for Maven 1.x

This page contains a list of the most current plugin releases. Check this page for updates with respect to the versions bundled in the official Maven release. See the main plugins page for other plugins.

Maven 2.0

Maven 2.0 is the current stable release of Maven. For more information and installation instructions, see the Maven 2.0 download page.

1.3.1 Release Notes

Maven 1.1-beta-3 Released

The Apache Maven team is pleased to announce the release of Maven 1.1-beta-3!

Download Maven 1.1-beta-3 and read the Installation Instructions.

Maven is a project management and project comprehension tool. Maven is based on the concept of a project object model: builds, documentation creation, site publication, and distribution publication are all controlled from the project object model. Maven also provides tools to create source metrics, change logs based directly on source repository, and source cross-references.

Compared to Maven 1.1 beta 2, this release focuses on the following objectives:

- Upgrade to later releases of dependencies, in particular Jelly, Dom4j and Jaxen,
- Re-introduction of Xerces into the core dependencies (it was removed in previous betas),
- Upgraded versions of almost all the bundled plugins,
- Documentation updates,
- Bugfixes.

For a full list of changes, please see JIRA.

With just a few exceptions, Maven 1.1-beta-3 is backwards compatible with Maven 1.0. One notable issue is the fact that you cannot use xml entities in project.xml, see MAVEN-1755.

Please note : Maven 2.0.4 is the latest stable release of Maven and is recommended for all new projects, but is not compatible with Maven 1.x. The release of Maven 1.1 is to improve the performance and stability for those using Maven 1.0 for their builds.

We hope you enjoy using Maven! If you have any questions, please consult:

- the FAQ
- the maven-user mailing list

- The Apache Maven Team

1.3.2 Compatibility

Compatibility Issues

This page details some of the backwards compatibility issues that you might encounter upgrading between Maven versions, and how to update your project. You'll find here:

- Changes Between Maven 1.0.2 and Maven 1.1: If you are using the last stable release.
- Changes Between Maven 1.1-RC1 and Maven 1.1: If you are using the last testing release.

Changes Between Maven 1.0.2 and Maven 1.1

JDK requirement is now 1.4 and above

The JDK required to *run* Maven is now 1.4 or above. This does not affect what JDKs you can build your libraries for, however - you can still generate class files that will run on 1.1 JVMs as before.

Bundled Ant version changed from 1.5.3-1 to 1.6.5

While in most cases this change should be transparent, those **using optional tasks will need to add them as a dependency to their plugin/project**, as not all optional tasks are bundled with Maven any more.

Deprecated Plugins Removed

The following plugins are no longer distributed with Maven by default. They are still available using the `plugin:download` command to obtain the last version released, and the source code is available from the plugin sandbox. If there is sufficient interest in maintaining them or moving them to another project it will be considered.

- abbot
- appserver
- ashkelon
- aspectwerkz
- caller
- castor
- docbook *(note that there is a superior docbook plugin at the SourceForge Maven Plugins project)*
- hibernate
- j2ee
- jboss *(can be replaced by Cargo)*
- jbuilder
- jcoverage
- jdee

- jdeveloper
- jetty *(can be replaced by Cargo)*
- jnlp
- junit-doclet
- latex
- latka
- release *(replaced by scm plugin)*
- repository
- shell
- struts
- tjdo
- uberjar *(can be replaced by javaapp)*
- vdoclet
- webserver
- wizard

The `default` attribute in maven.xml has been deprecated

The default goal should be specified in a `<defaultGoal>` tag inside the `<build>` section of project.xml instead of maven.xml.

Implicit dependencies on JARs in Maven's `lib` directory

The following libraries are no longer distributed with Maven:

- `commons-digester`
- `commons-graph`
- `commons-lang`
- `which`

If a plugin did not declare a dependency but relied on its existence, it may fail to work under Maven 1.1.

In addition, the versions of some libraries have changed, which may affect plugins.

Upgrading `maven.jar`

If you are directly importing and using classes from this package, you may find classes from the `org.apache.maven.project` package are missing as they have moved. This may also apply if you are using `maven-jelly-tags`.

Fix: Include `maven-model-3.0.1` in your project's dependencies.

Changes to `org.apache.maven.project.Repository` (and use of `pom.repository.*` in Jelly)

The `Repository` class now houses only the information from the project model. You may find that static methods such as `splitSCMConnection` and `tokenizerToArray` or fields such as `scmType`, which depended on the split function, are missing.

Fix: The static methods have been removed from the core but they exist in the `RepositoryUtils` class in `maven-changelog-plugin` if needed. All other plugins had been manually splitting the connection string. See `maven-scm-plugin` for an example of this. The future should see all plugins and code using Maven SCM for this task, and all dealings with an SCM system.

Use of `Project.getDependencyPath` and `Project.getDependency`

If you have a single non-JAR artifact that you resolve using this function, it will no longer be found.

Fix: Specify the type in the argument. The full form is `getDependency('groupId:artifactId:type')`. Type is defaulted to `jar`.

XML APIs

As in maven 1.0, xercesImpl,xml-apis and xml-resolver are loaded in the endorsed libraries of the JVM. You can't override them and if you use another version of xml-apis in your tests, you have to fork the JVM.

If in a custom goal or in a plugin, you use some xslt transformations, we recommend that you not use the style task in ant which will not work with all JDKs. Instead, you are encouraged to use a custom process launched in a forked JVM (like it's done in the PDF plugin for example).

Parse errors on previously 'valid' project files

`project.xml` files that used to work in older versions of Maven may now present parse errors. The parser in Maven 1.1 is much less tolerant of invalid `project.xml` files to avoid silently masking syntax errors.

Fix: Check your project file against the XSD published for the model version you are using.

Deprecated `create-upload-bundle` goal.

Replaced by `artifact:create-upload-bundle`.

New default repository

The default repository is now http://repo1.maven.org/maven/ (instead of http://www.ibiblio.org/maven/).

Issues in bundled plugins

Changes Between Maven 1.1-RC1 and Maven 1.1

Upgraded plugins

Include upgraded (bug-fix) versions of the test and artifact plugin.

1.3.3 Road Map

Roadmap

The current stable release for maven 1.x is version 1.1, obtainable from the download page. No further development is planned for the 1.0 branch, and 1.1 is in maintenance mode, i.e. development is restricted to bug fixes.

The list of unresolved issues / enhancement requests for Maven 1.x can be browsed under the MAVEN JIRA project.

Most development on Maven is now done on the 2.x branch. Please visit the main Maven web site for information on Maven 2.

Maven 1.1

Maven 1.1 has been released, you can download it here.

The main changes with respect to Maven 1.0.2 are:

- Integration of Maven 2 technologies such as Maven Wagon, Maven SCM and the new model code
- Ant 1.6 support
- Upgrade to later releases of dependencies, in particular Jelly, Dom4j and Jaxen
- Improved POM layout
- Updated versions of almost every bundled plugin
- Documentation improvements
- Bugfixes - check the JIRA changelog.

With just a few exceptions, Maven 1.1 is backwards compatible with Maven 1.0.

2.1 Getting Started

Getting Started

This guide will give you quick instructions on how to get Maven up and running on your system.

- Concepts - Concepts to understand before using Maven
- Installing - Installation instructions
- Building a Project - How to use Maven to build a given project
- Creating a Project - A 'Ten Minute Test' on how to create a project with Maven
- Adapting a Project - How to adapt an existing project to use Maven
- Migrating from Ant - How to migrate an existing Ant build to Maven

2.1.1 Installing

Installing Maven

You must first download Maven (please refer to the documentation).

Please refer to the appropriate section for the download you have obtained:

- Windows Installer
- Other Archives

The final optional step is to set up a `${user.home}/build.properties` file to customise your Maven installation. For information on the properties you can use, see the Properties Reference.

Windows Installer

Installing from the Windows Installer behaves like other Windows Installers: simply run the program and follow the prompts.

If you were updating a previous installation, you do not need to do anything further. Note that Maven does not overwrite a previous install, so you may want to remove it from *Add/Remove Programs*. This can be done safely at any time, as it does not affect your user profile.

During the process, you must select an installation directory. This will be set as `MAVEN_HOME` in your environment, however if you need to be able to run Maven from anywhere, you should add it to your path. To do this under Windows 2000 and Windows XP, open the Control Panel, and open the System panel. Under the Advanced tab, select the Environment Variables button. Create a new user variable (or edit it if it exists) to add `%MAVEN_HOME%\bin` (eg. `PATH=%PATH%;%MAVEN_HOME%\bin`). **Note :** Since Maven 1.1 RC1 the installer automatically adds `%MAVEN_HOME%\bin` in the path.

You will also need to define the `JAVA_HOME` environment variable. This variable should be the directory where a Java Development Kit is installed (note that a JRE is not sufficient). This directory will contain the `bin`, `jre` and `lib` directories.

Next, you should create your local repository by running the following command:

```
For Windows:
  %MAVEN_HOME%\bin\install_repo.bat %USERPROFILE%\.maven\repository
```

Notes:

- This step is optional, but will save downloading several JARs a second time.
- On Windows systems, your local repository is set by default to `%USERPROFILE%\.maven\repository`, which can be annoying when using roaming profiles, you can change its location using the maven.repo.local property.

To confirm that you can start Maven, run

```
maven -v
```

Other Archives

If you have downloaded an install archive, it contains a single top-level directory named `maven-VERSION`, where VERSION is the version downloaded (eg. 1.0). This directory contains all the Maven related files beneath that. You can now unpack the install archive using `tar` or `unzip`.

Before you begin using Maven you will need to

- define the `MAVEN_HOME` environment variable which is the directory where you just unpacked the Maven install archive,
- add `MAVEN_HOME/bin` to your path so that you can run the scripts provided with Maven.

If you are updating from a previous version, you do not need to do anything further. You can safely remove the old install if you wish as it does not contain any user profile information.

You will also need to define the `JAVA_HOME` environment variable. This variable should be the directory where a Java Development Kit is installed (note that a JRE is not sufficient). This directory will contain the `bin`, `jre` and `lib` directories.

Next, you should create your local repository by running the following command:

```
For Unix:
  $MAVEN_HOME/bin/install_repo.sh $HOME/.maven/repository
```

Note: This step is optional, but will save downloading several JARs a second time.

To confirm that you can start Maven, run

```
maven -v
```

2.1.2 Building a Project

Quick Start - Building a Project with Maven

Ok, so you've downloaded a project that requires Maven to build, and you've downloaded Maven and installed it. This guide will show you how to use it on that project.

If you find you don't understand the meaning of a term, you should refer to the Glossary for commonly used terminology that is either unique to Maven or has a special meaning.

Being Prepared

Maven will run with sensible defaults, so you can get right into it. However, if you are operating under a particularly restrictive environment you might need to prepare to run Maven. When Maven first runs it will:

1. Expand plugins into a cache directory
2. Download additional dependencies for the plugins and project being built

The Properties Reference explains how to configure Maven to use different locations for the cache and download repository. By default, these reside under `$HOME/.maven`.

The first time a project is built or a specific plugin is used, you may need to be online to obtain dependencies. If you have them, JARs can be put into your local repository manually if desired, and you can also run a local mirror of needed dependencies. For more information, refer to Working with Repositories.

This might seem onerous, but remember the following advantages:

- All Maven projects (including Maven and its plugins) share the repository, so each dependency is only downloaded once.
- This avoids needing to download each dependency manually and place it in the correct place for each build.
- No large binaries need to be stored in CVS where they are downloaded and checked for updates often.
- Maven automatically and uniformly handles proxy servers, multiple remote sources and mirrors.

Occasionally, some dependencies will not be downloadable because of the distribution license they have. These must be downloaded manually and placed into the local repository. This is explained for the commonly requested Sun reference JARs. See Standard Sun JAR names for more information.

Common Tasks

Maven provides standard tasks for common operations. For example, to build a JAR from the current project, run:

```
maven jar
```

You'll notice that all of the code is compiled, the unit tests are run, and finally it is packaged into a JAR file. The unit tests generate reports that can later be parsed to create an HTML report that is part of the generated site documentation. The JAR file can also be automatically distributed if making a release, or shared with other local projects.

The output of this task is in the `target` directory, as with all Maven output. For example:

- `target/classes` - compiled source classes and resources
- `target/test-classes` - compiled test classes and resources
- `target/test-reports` - Text and XML representations of the test results
- `target/project-X.Y.Z.jar` - the output JAR, built from the `target/classes` directory.

For more information on the JAR plugin, including references for goals and properties, see the jar plugin documentation.

The individual goals that make up the JAR goal can also be called individually, for example:

```
maven java:compile
maven test
```

If you are not building a JAR, there are other plugins for building different types of artifacts. Examples of these can be seen from the "Using Maven" section in the left hand navigation.

Another common task when building from the source is to build a local copy of the project site for documentation and reference. This is done with the goal:

```
maven site
```

Note: building the site can require a large number of dependencies to be downloaded the first time as several different reports are run.

The result of this command is a `target/docs` directory in the project's base directory that contains an entire web site of documentation. If you are the website maintainer, you can also publish this to a local or remote web server.

As for any plugin, more information on the `site` goals can be seen in the site plugin documentation. In addition to a goal and property reference, some plugins also have their own set of FAQs and examples.

To clean up after a build and start fresh, simply run:

```
maven clean
```

While the standard goals suffice for most day to day usage of Maven, there are plenty of other plugins to use. The next section will explain how to get specific help building a project.

For information on incorporating Maven into your own project, start with the Ten Minute Test.

Getting Help on a Maven Project

Ok, you have a project you've just downloaded, and you know it builds with Maven. There's no documentation to speak of for the project itself, so where to start?

Help in Ant

If you were using Ant, you'd probably run `ant -projecthelp`, and if that didn't help, just run "ant" and hope the default target is what you wanted. A `build.properties.sample` file may have been provided so you'd copy that and edit it to see if that helped. Then you'd fix individual problems as they came up. If everything goes wrong and you can't find the right target or what property it's expecting, you dig into `build.xml` and look at what it is attempting.

The process for Maven should be at least as easy, and usually simpler.

Help for a Maven project without a maven.xml file

If there is no `maven.xml` file, then you know the build hasn't been customised so you can rely on a few things working.

If you know it is a JAR, `maven jar` should do what you expect. If you know it is a web application then `maven war` should produce the expected results, and if there are subprojects, `maven multiproject:artifact` is the standard build task. Finally, `maven site` (or `maven multiproject:site`) should produce some documentation or at least the standard reports.

Though it is much less likely than in Ant, a `build.properties.sample` file may have been provided so you'd copy that and edit it too.

Otherwise, you shouldn't expect to have to define any properties outside what the project has provided and you have already defined to get Maven running, as only the standard Maven plugin properties are being used, and you know how they behave.

Help for a Maven project with a maven.xml file

If there is a `maven.xml` file, then the build has been customised. You can still rely on the above goals doing what they advertise, but they might not be the recommended way of building that particular project. This is especially true when projects define their own multi-project structure without using the multiproject plugin to build.

The first step would be to run `maven -u`. Similar to `ant -projecthelp`, it will list all of the goals in `maven.xml` and any description listed with the goal by the author. Standard goals like `jar` will be omitted as their behaviour is already known.

You may just run `maven` and rely on the default goal being what you wanted.

You may still get some of the ant problems, where the above doesn't help or the properties are required and not defined. The same process will apply here - you investigate `maven.xml` which should be much shorter as it only contains customisations.

There will always be bad builds under either system, but Maven gives you quite a head start by providing a standard set of build pieces that you can expect to work out of the box.

Where it usually goes wrong in Maven

Having said that, there are still a few areas where a Maven build won't work out of the box. By far the most common occurrence of this is depending on a JAR that is not published to a public place. This might be because of licensing (eg a lot of the standard Sun JARs cannot be redistributed by Maven), or because it is a development build of some library that was being built locally. For more information, see Standard Sun JAR names.

If you are downloading the source to a project, it is always worth running `maven build:end` once on each `project.xml`. This won't do anything except download all the dependencies the project wants. If some can't be found, then you should track them down while still online. If it has been particularly troublesome and undocumented, you might want to harass the authors of the project via their users mailing list (which you should also find in `project.xml`).

More Help

For a complete reference on how to get help about building a Maven project from the command line, see the Maven Command Line Reference.

2.1.3 Creating a Project

The Ten Minute Test - Creating a Project with Maven 1.x

Often when evaluating a new piece of software, you want to give it the "ten minute test" - download, install and run it, and then do something useful with it within 10 minutes to get a feel for how it works. It can be a lot more useful than reading the manual. This guide will help you do that.

Sure, you won't have converted any massive, highly-customised build you may already be using, but you'll know where to start.

Beforehand, it is worth getting familiar with how to run Maven. Please read the Quick Start for important instructions on how to install Maven, and download dependencies, and learn the basics for running it.

If you are already very familiar with Ant, you may also like to refer to the Migrating from Ant guide.

Step 1: Setting up the directory structure

As you'd expect, the first steps are to set up the required directories. The small project will be an echo program.

```
mkdir sample-echo
cd sample-echo
```

From here, you *could* use the genapp plugin to generate a skeleton project for you. However, here we will start from scratch so that each element can be explained.

While you can set up almost any directory structure and use it with Maven, there is a standard directory structure defined that is recommended. The reasons for this are explained in the definition, but essentially ensure minimal extra configuration by you, and make it easier for users familiar with Maven to navigate your project.

Let's continue by creating the following directory layout:

```
sample-echo [current directory]
+- src
   +- main
      +- java
         +- samples [package structure]
            +- echo
      +- resources
   +- test
      +- java
         +- samples [package structure]
            +- echo
+- xdocs
```

The following is an explanation of what each directory's purpose is:

`src`	This will contain all types of source files under the following subdirectories.
`src/main`	This is intended for the main part of the application - the code that would be part of the final distributable.
`src/main/java`	This is the java code that makes up the application. Below this should be a package structure.
`src/main/resources`	These are additional resources for copying into the final distributable. This may have a subdirectory structure that is maintained, including using a package structure. For example, you might have a `META-INF/MANIFEST.MF` file in here (although Maven does create a default one for you so this isn't usually necessary).
`src/test`	This contains everything needed to unit test your application. You may have additional similar directories later if you add other types of tests such as integration tests using Cactus.
`src/test/java`	This is the java code that makes up the unit tests for the application. Below this should be a package structure.
`src/test/resources`	As for `src/main/resources`, but only made available to the unit tests. Not used in this example.
`xdocs`	This contains the documentation that will be transformed into HTML and published as a project site. **Note**: by defaulting the xdocs location to the top level directory, Maven 1.x violates the directory structure conventions adopted in Maven 2, where it defaults to `src/site/xdoc`. You can make your project layout compatible with Maven 2 by overriding the `maven.docs.src` property.

Now, you are ready to start defining the project layout for Maven.

Step 2: Create a project descriptor

The most important file to Maven is `project.xml`. While you can run Maven without it, it will not know anything about your project - so is only useful for project-independant goals such as `scm:checkout` or `genapp`.

Different people have different preferences for how they create their project descriptor. Some will copy from an existing project and edit, some will run `maven genapp` and accept the defaults to get a skeleton project, and some will hand-edit it from scratch. In the near future, tools will be available to graphically edit the project descriptor.

The following is a basic project descriptor:

```
<project xmlns="http://maven.apache.org/POM/3.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/3.0.0
http://maven.apache.org/maven-v3_0_0.xsd">
  <pomVersion>3</pomVersion>
  <groupId>sample</groupId>
  <artifactId>sample-echo</artifactId>
  <name>Sample</name>
  <currentVersion>1.0-SNAPSHOT</currentVersion>
  <inceptionYear>2005</inceptionYear>
  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.8</version>
    </dependency>
  </dependencies>
  <build>
    <sourceDirectory>src/main/java</sourceDirectory>
    <unitTestSourceDirectory>src/test/java</unitTestSourceDirectory>
    <resources>
      <resource>
```

```
        <directory>src/main/resources</directory>
      </resource>
    </resources>
    <unitTest>
      <includes>
        <include>**/*Test.java</include>
      </includes>
    </unitTest>
  </build>
</project>
```

There are many more elements to be introduced as you develop, especially if you want to make releases or publish a site. A full description of the project descriptor is available. Often, common elements will be shared in a parent descriptor to keep each as small as possible.

The above descriptor is all you need to build a JAR and run any tests associated with it. Try it for yourself: add some Java code to the `src/main/java` directory (include subdirectories for any package name), and a JUnit test under `src/test/java`. The following goals will perform some standard behaviours:

- `maven java:compile` - this will compile the code and check for errors - nothing more
- `maven test` - this will compile the code and tests, then run all of the unit tests
- `maven jar` - this will build a JAR from your code, after running the tests as above
- `maven site` - even now, you can generate a site in `target/docs` and see what it will look like

Note that you can download some sample code instead of creating the project above.

Customising the Build

In addition to the project descriptor, a number of properties can be set to customise the way the build performs. For example, say you wanted to change some properties regarding compilation for the project. You can create a `project.properties` file in the same location as your `project.xml` file.

```
maven.compile.source=1.3
maven.compile.target=1.1
maven.compile.debug=true
```

There are a great number of customisation properties available, each relating to individual plugins and some for Maven itself. For example, the above properties are defined for the Java plugin which provides the `java:compile` goal.

You can usually find a reference of the properties that will customise a plugin in each respective plugin's documentation. A list of plugins can be found in the Plugins Reference, which is also linked in the navigation of this site.

There are also a number of standard properties defined by Maven itself, which are listed in the properties reference.

Properties are read in a particular order - while usually they will be defined in `project.properties` to

share with all users of your project, sometimes they are specific to your environment (eg, when they include paths on your machine). In this case they should be included in `build.properties` in either the project directory (if they are specific to the project) or in your home directory (if they are for every project on your machine under your account). For more information about the order properties are loaded, see the properties reference.

Generating the Site

As was shown previously, the project descriptor is all that is initially needed to generate a site for your project by running:

```
maven site
```

The next step in this process is to add some custom documentation to the site. This might be some basic usage instructions, linking out the Javadoc and other references, or it might be a complete project site depending on your preference.

To try this, add a file called `xdocs/index.xml` in your project with the following content:

```
<document>
  <properties>
    <title>Hello World</title>
    <author email="me@mycompany.com">Me</author>
  </properties>

  <body>
    <section name="Section 1">
      <p>
        Hello world!
      </p>
    </section>
  </body>
</document>
```

To view your new index page in `target/docs/index.html`, regenerate the site again:

```
maven site
```

Any XHTML will be accepted within the <section /> tags, as well as other special tags. Some HTML, such as tables, are also styled specially. For more information on building a site, see the User's Guide.

Where to next?

Thanks for giving Maven the ten minute test! If you have any feedback (positive and negative), please email the Maven Users List.

This tutorial has hopefully given enough information on how easy it is to start a new project with Maven. Luckily, creating more complex projects is just as easy as Maven encourages you to build up projects

consistently, like building blocks.

For more information on beginning with Maven, the book Maven: A Developer's Notebook has a sample chapter online that deals with getting started.

It is recommended that you read the areas of the User's Guide that relate to the particular type of project you are working on next. This includes:

- Utilising source control
- Working with multi-module projects
- Releasing a project and deploying it to a repository

As previously mentioned, you can also find more help on using a particular plugin in each plugin's documentation mini-site, which are listed in the Plugins Reference. There is also a Reference area for more general Maven documentation.

2.1.4 Adapting a Project

Adapting an Existing Project to Use Maven

This guide will discuss introducing Maven to an existing project which may have some other build system or none at all. It is assumed that you are now a little familiar with Maven, and have given it the Ten Minute Test, so have already created a simple project.

There is really only one step to introducing Maven into a project: the creation of the project descriptor. Since this does not affect your other build files and should be able to match your current layout, the process is the same as for starting a new project, however more of the values will be customised.

This approach can allow you to start getting some of the benefits of Maven without disturbing something that is already working.

However, eventually this will mean maintaining information in two locations, which is not ideal. Hopefully you will see that Maven offers the functionality of your existing build environment without much additional work and so will migrate towards using that as the sole platform.

For Ant users, there is a document called Migrating from Ant that will be of assistance.

No matter what you are using, the principles are similar: move small portions of your build to Maven, and make Maven the central entry point for goals. For functionality remainining in the existing environment, Maven can call out to it to perform those tasks, using `<ant:ant/>` or `<ant:exec/>`.

While Maven can usually be configured to match your environment, it is also important to consider refactoring your build, in the same way you would for your code. Changes small, known pieces at a time towards a layout that is encouraged by Maven's Conventions Reference.

If you need real-world examples, there are plenty of Maven powered projects that can be used as references. However, it is worth checking more than one and referring to the Best Practices document as you do this.

2.1.5 Migrating from Ant

Migrating from Ant

A common question is how to migrate an existing Ant build to Maven. The first section of this guide will help in some small ways to do that. It will not touch on the reasons for using Maven over Ant, or whether Maven or Ant is better for your project - these issues are discussed below in Maven for Ant Users.

If you are looking for some tips on how to do certain things you may have been doing in Ant, see the relevant section of the FAQ.

Getting Started - Creating a Project Descriptor

Sometimes people want to get their feet wet with Maven by using it to do the standard things, but to use their existing Ant build that is working fine for complex tasks. This is possible.

To do anything with Maven, you will still need to set up a project file with the information you need to do the Maven tasks you are using.

Even when you are already using Ant, this project file can be set up like any other new project. For a tutorial on doing so, see the Getting Started guide to creating a project.

Using Existing Libraries

Often, an Ant project will look for dependent libraries to add to the classpath in a known location on the filysystem (often a subdirectory of the build checked out of source control along with the source code). In some cases, the `<get/>` task is used to retrieve them remotely.

It is recommended that you establish repositories and let Maven take care of this. If the dependencies are not in the Maven central repository, set up an internal repository. Publish your resulting artifacts to that repository also.

However, if there is a necessity to maintain the file based approach, or you wish to start with that to keep the number of changes down, you can use JAR overrides to have Maven look for dependencies on the file system. For more information, see Handling Dependencies.

Calling Ant Scripts from Maven

In the same directory as this `project.xml` file (and probably in the same place as your ant build) you should create a `maven.xml` file like so:

```
<project xmlns:ant="jelly:ant">
  <goal name="do-ant-bit">
    <ant:ant dir="${basedir}" antfile="build.xml" />
  </goal>
</project>
```

You could then run `maven do-ant-bit` to run the old ant script. Ok, so this isn't that useful yet, but it can be used in conjunction with hooks to start merging an old Ant and new Maven build.

For example, consider you are now building your site with Maven. However you have an old part of your Ant build that generated a few HTML pages that you want copied with the site. Here is a possible solution.

```
<project xmlns:ant="jelly:ant">
  <preGoal name="site">
    <!-- Set the property html.dir that your Ant build uses to put the HTML into -->
    <ant:property name="html.dir" value="${maven.docs.dest}" />
    <ant:ant dir="${basedir}" antfile="build.xml" target="generate-html" />
  </preGoal>
</project>
```

Building on your Maven Project

Once these fundamentals are in place, the next steps of integration would be to start importing small Ant fragments from your existing build into your `maven.xml`, or for larger tasks make them reusable and create a Maven plugin that will be available to all your projects.

Some tasks may already have a complete equivalent in Maven (eg, creating a JAR), so you can remove that section from the ant script altogether.

The easiest way to use the Maven features while still executing the existing build at the appropriate time is to use goal hooks like the one given in the previous section. For more information on these, see Customising Goals in the Scripting Reference.

After this, your build will continue to grow like any other Maven build would. For information on how to do this, see Customising Maven and the Scripting Reference, as well as other sections in this User's Guide.

Maven for Ant Users

This section is intended to answer the commons questions of someone that is familiar with Ant, and wants to learn what the equivalent concepts and the differences in Maven are, and where each product excels.

While there is a strong slant towards Maven, this is NOT a guide designed to bash Ant, a remarkable and fine product. This guide IS for helping an Ant user understand the remarkable and powerful capabilities within Maven.

Different Objectives

It is important to realise that Maven and Ant have different objectives. For clarification, it is worth reading What is Maven? to learn what Maven is, and what Maven is not.

You may have heard that Maven is just Ant plus dependencies, or a set of reusable Ant scripted plugins.

But in fact, the aims of the two products are quite different. There are other solutions that bring reusable fragments and dependencies to Ant 1.6 - but these do not make Maven redundant because, aside from the many other features Maven also has, it actually takes a different approach.

Build Knowledge

One of the fundamental differences between Ant and Maven is in whose responsibility it is to understand the build process and the tools used therein.

With Ant, it's the developer who must understand what the tool is and how it applies to their development. With Maven, build process knowledge is captured in `plugins`, small snippets of processing that rely on you providing some information.

For example, to compile your java code using Ant, you must first write a `build.xml` file which uses Ant's javac task, passing that task the correct set of parameters.

```
...
<!-- compile Java source files -->
<target name="compile">
  <javac srcdir="./src;./test" destdir="./bin" debug="on" optimize="off">
    <classpath>
      <fileset dir="lib" includes="*.jar"/>
    </classpath>
  </javac>
</target>
...
```

You'd then call this `build.xml` file using ant from the command prompt:

```
ant compile
```

This isn't too hard to do, and as you move from project to project you become better at this, and understand Ant better, learning what each of the options are for and what the best way to write these snippets is.

Maven, however, takes a far more declarative approach. You must provide Maven with some information about your project (in a file called `project.xml`), one piece of which is your source directory that stores your java code.

```
...
<build>
  <sourceDirectory>src/java</sourceDirectory>
...
```

To compile your code, you just need to specify that you have a source directory and then ask Maven to compile using:

```
maven java:compile
```

There's no need for the developer to understand how to write a plugin to use one. And several plugins can share the same information. For example the plugin that checks that source code conforms to a coding standard uses the same `sourceDirectory` as the plugin that compiles code.

This abstraction also ensures consistency. No matter what project you are building, `java:compile` will compile the source code, the source code is easy to find, and the output is always in the same place (`target/classes`).

Best Practices

Another key difference in philosophy is that of best practices. Ant is a very flexible tool - because you can break your build down into targets, which is further composed of discrete tasks you can combine them in many ways to build your project.

While Maven still allows this flexibility, because it is built around a standard set of functionality and uses metadata with sensible defaults to the describe a project, it attempts to enforce the use of best practices in composing the project's build. These best practices have been built up from the collective experience within the community.

Some examples of what Maven does in this regard are:

- Standard locations for source files, documentation, and output
- A common layout for project documentation to make finding information easier
- Retrieves project dependencies from a shared storage area, instead of keeping multiple copies of a JAR in CVS
- Encourages the use of a single source directory, but a separate unit test source directory

More than the Build - Project Knowledge

Maven also aims to go beyond the building of a project from sources, and encapsulte all of the project knowledge in one area. This includes the source code and the documentation, but also includes reports and statistics on the code and the processes surrounding management of the code.

Maven not only handles building the code, but can handle the release process, integrating closely with source control to get a clean copy to build before releasing a new version that is also deployed for others to use.

Maven is able to integrate with a large number of different products such as issue tracking, source control systems, reporting systems, application servers and software development kits. While Ant also has (and provides) much of this integration, Maven continues to use its declarative style of set up that means there is less and sometimes no work to set up these new integrations.

Terminology

This section covers what different components are called in each of Maven and Ant.

Project

In Ant a *project* is what you define in your `build.xml` file. This project is a collection of *targets*, which are composed of *tasks*.

In Maven a project is what you define in your `project.xml` file. It's a collection of information about the development effort that Maven uses to produce an artifact such as a JAR or WAR. Only a single artifact is built in a project: if you have multiple components to a project you build them together as a set.

Target

A *target* is the unit of execution for Ant. When you execute Ant, you provide a list of targets to run, e.g.

```
ant clean compile
```

This executes the `clean` target followed by the `compile` target that you have coded in your `build.xml` file.

In Maven, the *goal* is the equivalent of a target, e.g.

```
maven clean java:compile
```

This executes the `clean` goal of the `clean` plugin, followed by the `java:compile` goal of the `java` plugin.

Maven goals can be pre-defined in plugins as above, or you can define your own in a `maven.xml` file. In Maven 1.x, goals are written in Jelly, and can utilise any Ant tags from the version of Ant provided with Maven.

Build file

When writing your own code in Ant, you place the targets in the `build.xml` file.

When writing your own code in Maven, you place the `goal`s in the `maven.xml` file. However, if you are only using standard Maven goals, you may not even need a `maven.xml` file.

2.2 Using Maven

Using Maven

This guide will describe how to work with Maven to build different types of projects. If you haven't already, it is recommended that you first read the "Getting Started" documents found in the navigation of this site for an introduction to working with Maven.

- Building JARs - How to build an individual library
- Resources - Adding more files to the generated library
- Unit Testing - Running unit tests
- Creating a Site - Adding to the project web site
- Source Control - Working with SCMs
- Making Releases - How to cut a release
- Web Applications - Working with web applications
- Multiple Modules - Building multiple libraries and applications together
- Handling Dependencies - Managing the dependencies of your build
- Migrating from Ant - Where to start if you already have `build.xml`
- Best Practices - Why Maven encourages what it does, and how to take advantage of it
- IDE Integration - How to use Maven with your IDE

Further Reading

Further topics in the User's Guide include:

- Customising Maven - how to change behaviour, add scripting and create your own plugins
- Repositories - how to manage your own artifact repository
- Reference - some quick reference documents, in particular Maven Conventions and a Command Line Reference.
- Plugins - describes the various plugins available for Maven 1
- Contributing - what to do if you want to help us make Maven even better!

2.2.1 Building JARs

Building JARs

Building a JAR with Maven is one of the simplest uses, and it is the default project that Maven will build.

For a basic build example, you can see the Getting Started Guide. This goes through the steps of setting up a simple Java project and building a JAR.

What will happen is that the Java code will be compiled (as specified by the source directory you've given), resources copied into the root location for the JAR (also given in the project descriptor), the tests run, and a JAR bundled up.

To add additional files, such as a manifest - you can add more resources, and they will be copied into the root of the JAR with directories intact.

If you would like to customise the existing manifest, there are several properties that can be set. For example, `maven.jar.mainclass=com.mycompany.Main` will set the `Main-Class` attribute.

Everything you do with Java in Maven will build from this foundation. The following sections will describe resources, tests, and other functionality that is available to build files other than JARs.

For more information on the specifics of the JAR, please consult the JAR plugin documentation. Goals in Maven will have their documentation listed under their respective plugins in that documentation reference.

To continue to the next topic, see Working with Resources.

2.2.2 Resources

Resources

Resources are used to include additional files in the build, or use additional files for testing. They are copied into the classpath before compilation.

Resources can be specified in two places: the `build` element (reference) and the `unitTest` element (reference). These resources are specified identically, but are used separately - the first for the main build (which is also given to the unit tests), and the other only for the unit tests.

The following is a simple use of resources:

```
<resources>
  <resource>
    <directory>src/main/resources</directory>
  </resource>
</resources>
```

In this example, every file in `src/main/resources` is copied as is to the `target/classes` directory, preserving any subdirectory structure.

Pattern sets can be used to include and exclude certain files, for example:

```
<resource>
    <directory>src/main/resources</directory>
    <includes>
      <include>**/*.xml</include>
    </includes>
    <excludes>
      <exclude>directory1/dummy.xml</exclude>
    </excludes>
  </resource>
```

In some cases, a specific resource might need to be copied to a particular subdirectory. Usually, this just means having the same subdirectory structure in your resources directory, but another alternative is to use `targetPath`:

```
<resource>
  <directory>src/main/meta</directory>
  <targetPath>META-INF</targetPath>
  <includes>
    <include>MANIFEST.MF</include>
  </includes>
</resource>
```

Resources can also be filtered for tokens like `@property.name@`, identically to Ant. First, you must enable filtering for your resources.

```
<resource>
  <directory>src/main/resources</directory>
  <filtering>true</filtering>
</resource>
```

At the moment, you must define Ant filters to achieve this. This can be done using a `preGoal` on `java:jar-resources`, for example:

```
<preGoal name="java:jar-resources">
<ant:filter token="some.property" value="some_value" />
<ant:filter filtersfile="some.properties" />
</preGoal>
```

Note however that filters may cause issues with keeping a single build reproducible. Please see the Best Practices document for more information.

2.2.3 Unit Testing

Unit Testing

The Maven team believes that unit testing can be a very useful tool in improving the quality of software, in particular to enforce a loosely coupled design and to give the ability to regression test the code over time. For this reason, unit test support (provided by JUnit in Java) is included out of the box, and an integral part of the build process.

To configure your test suite, add the `unitTest` element to the project descriptor.

By default, the tests are run whenever you package your artifact to ensure they continue to pass. They can also be run explicitly using the command: `maven test:test`.

If you need to skip tests temporarily for speed, you can use the `maven.test.skip` property:

```
maven -Dmaven.test.skip=true jar
```

This is not generally recommended, however - your tests should remain fast enough that running them regularly does not have a negative impact.

There are other test goals to run individual tests or a subset of tests, as well different configuration options. For more information, see the test plugin reference.

2.2.4 Creating a Site

Creating a Site

Maven can be used to generate an entire web site for your project. The web site can be used to house all of your user-facing documentation, generated as flat HTML with a common look and feel.

It also contains a number of generated documents based on the information in your project. This can include SCM information, mailing lists and a list of developers, as well as FAQs and release notes based on additional metadata you provide.

Finally, the web site will contain various reports about your source code generated by the plugins included in Maven such as javadocs, metrics, unit test results, changelogs, and more.

Adding Documents to the Site

Creating a Site Descriptor

To add documents to the site, first you will need to add them to the navigation so that they can be found.

To do this, create a `navigation.xml` file in the `xdocs` directory that contains the menu structure for the entire document tree. Add all items and sub-items.

For items that can collapse when you are not browsing them, add `collapse="true"` to the item declaration.

The following is an example `navigation.xml` file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<project name="Maven xdoc Plugin">

  <!-- Page Title -->
  <title>Maven xdoc Plugin</title>

  <body>

    <!-- Links defined across the top right of the site -->
    <links>
      <item name="Maven"  href="http://maven.apache.org/"/>
    </links>

    <!-- Menu in the Left Navigation -->
    <menu name="Menu Name">
      <!-- Standalone item -->
      <item name="Item" href="/item.html" />

      <!-- Uncollapsed Submenu -->
      <item name="Alpha" href="/alpha/index.html">
        <item name="Alpha One" href="/alpha/one/index.html"/>
        <item name="Alpha Two" href="/alpha/two/index.html"/>
      </item>
```

```
      <!-- Collapsed Submenu -->
      <item name="Beta" href="/beta/index.html" collapse="true">
        <item name="Beta One" href="/beta/one/index.html" collapse="true"/>
        <item name="Beta Two" href="/beta/two/index.html" collapse="true"/>
      </item>
    </menu>
  </body>
</project>
```

Note: the `href` element should be absolute, referenced according to your document root (ie, under `xdocs`).

You can also add images to items in the menu, for example:

```
<item name="The Site" href="http://www.thesite.net/"
img="http://www.thesite.net/thesite.png"/>
```

Creating a new Document

The XDoc format is quite simple, and is based on the original Anakia format that originated in Jakarta. The following is a sample XDoc document:

```
<document>
  <properties>
    <author email="user@company.com">John Doe</author>
    <title>Page Title</title>
  </properties>
  <!-- Optional HEAD element, which is copied as is into the XHTML <head> element
-->
  <head>
    <meta ... />
  </head>
  <body>
    <!-- The body of the document contains a number of sections -->
    <section name="section 1">

      <!-- Within sections, any XHTML can be used -->
      <p>Hi!</p>

      <!-- in addition to XHTML, any number of subsections can be within a section
-->
      <subsection name="subsection 1">
        <p>Subsection!</p>
      </subsection>
    </section>

    <section name="other section">

      <!-- You can also include preformatted source blocks in the document -->
      <source>
code line 1
code line 2
      </source>
```

```
    </section>
  </body>
</document>
```

To create one of these documents, create an XML file in the directory structure you want for your site under the `xdocs` document root. For example, `/start/introduction.xml` will be generated as `/start/introduction.html` on the final site.

Site Appearance

The following sections give some details on specific customisations you might want to make to the generated site.

For more information on customising the generated site, see the XDoc plugin reference.

Creating Your Own Index Page

Maven will generate a default `index.html` page for your project based on the `description` element in the POM. However, this is usually not very verbose, so you may like to create your own index page.

To do this, simply create an `index.xml` document in the base `xdocs` directory in the same style as the other documents (as explained in the previous sections), and this will be used instead of the generated document.

Visual Style

By default, all Maven-generated sites use the same look and feel for consistency. This enables users to easily recognize a Maven-generated site and immediately feel comfortable searching for information on the site.

Of course, we understand that users will want to customize their own sites. There are two alternatives.

Firstly, you can select different skins. Currently Maven comes with two skins - the current default and the "classic" skin that appeared in Maven 1.0 Beta 10 and earlier. Selection of the skin (and the colours used if using the classic skin) is done through properties given to the XDoc plugin.

The other alternative is to provide your own CSS. The Maven site is generated in XHTML and the layout defined by CSS. The best alternative is to take the existing stylesheet and modify it according to your tastes. To specify the alternate stylesheet, set the `maven.xdoc.theme.url` property for the XDoc plugin (again, see the properties for more information).

Reports

Finally, if you want to customize which reports are automatically generated for your site, you need to include the `reports` tag in your project descriptor. This tag enables you to selectively choose which reports to include and exclude as well as the order of the reports.

The following are the default set of reports, as specified by the XDoc plugin:

```
maven-changelog-plugin
maven-changes-plugin
maven-checkstyle-plugin
maven-developer-activity-plugin
maven-file-activity-plugin
maven-javadoc-plugin
maven-jdepend-plugin
maven-junit-report-plugin
maven-jxr-plugin
maven-license-plugin
maven-linkcheck-plugin
maven-pmd-plugin
maven-tasklist-plugin
```

Exclusion of All Maven-Generated Content

Users building documentation-only sites or sites that aggregate a lot of sub-projects might want to prevent the inclusion of the entire "Project Documentation" section of the navigation bar on their sites. This can be done by setting the `maven.xdoc.includeProjectDocumentation` property to the value of `no`. By default, Maven will include the "Project Documentation" section of the navigation bar which includes information about the project as well as links to the numerous Maven-generated reports.

2.2.5 Source Control

Working with Source Control

Working with your source control system in Maven is quite simple. Maven can provide the following services:

- Common tasks such as checking out code, tagging the current code, and performing an update
- Generating reports on the recent usage history of source control
- Providing links to a web view of the source control, and if desired publishing that information to the project site

Support is provided by the Maven SCM subproject. This provides support for the following SCM systems:

- CVS
- Subversion
- ClearCase (partial)
- Perforce (partial)
- StarTeam (partial)
- Visual SourceSafe (planned)

Configuration is extremely simple - in most cases, you need only specify the `<connection/>` string in the `<repository/>` element of your POM. Specifics about this connection string can be found in the SCM plugin documentation.

Performing Common Tasks

It is possible to perform several common tasks with Maven. These can be particularly useful for automation in your development environment, or in continuous integration.

There are also goals that provide assistance with making releases from your SCM.

For more information about common tasks and SCM based releases, see the SCM Plugin Documentation. This contains a complete guide to the `scm:` goals and related properties.

For general information on making releases with Maven, please see Making Releases.

Generating Reports

Maven provides three different reports on SCM usage that can be included in your project site. These are provided by the Change Log, File Activity and Developer Activity plugins.

These are included in your project site by default, however if you would like to explicitly select them for your project, add any or all of the following lines to `project.xml` inside the `<reports/>` section:

```
<report>maven-changelog-plugin</report>
<report>maven-file-activity-plugin</report>
```

```
<report>maven-developer-activity-plugin</report>
```

For examples of these reports, please see the ones published for this project: Change Log, File Activity, and Developer Activity.

There are other SCM reports available for Maven, such as the excellent StatCVS plugin. These are included on the 3rd Party Plugins list.

Adding SCM Information to Your Site

The source control information for the project is included by default in the project site. For example, see the SCM Usage page for this project.

For more information, please see Creating a Site.

2.2.6 Making Releases

Making Releases

Maven can simplify several areas of the release process for software but it is not a substitute for planning. Think about the process before starting any release.

Determine Format

Firstly, you must decide what form your distribution will take. For example:

- Standalone library artifact
- Binary and source distributions
- Installer

Once you have decided: it's time to set up maven.

If you plan to distribute your normal artifact as is then use the existing deployment goals. See Deploying to the Internal Repository for discussions about distributing raw artifacts. Public releases (of course) require deployment to a public repository. If your usual internal repository is private (and you want to use maven to upload the artifacts) then some changes to your project's properties may be needed for the release.

Distributions as binary and source archives are supported by the distribution plugin. For installers, currently, Maven only supports the NSIS installer on Windows. See the NSIS plugin for information. In both cases, some configuration will be necessary. You may have to add extra scripting to get the release just as you want it.

It may take a little effort to create and test release generation but it only needs to be done once. Maven will then be able to reliable generate all future releases in this format.

Check Dependencies

A key step towards making a release in Maven is to verify that your build is going to be reproducible (see Best Practices for more).

You should check the dependencies of the project you will be releasing. Ensure that all dependencies on snapshots or other non-final releases are resolved to proper releases. Then test that the software runs correct against them.

Releasing against a development dependency may lead to problems later. Not only may this cause future compatibility problems for your users but if you remain pointing to a development dependency that is later overwritten, attempting to rebuild your release may lead to at best a different result, at worst a failure to build.

If there is no release of the software, but you know the current development build to be stable, as a last resort you should internally release that library (using the current time as a version, or similar). At least then your software has a concrete dependency that should not be overwritten.

Future Maven release tools will help to identify the existence of such snapshots and help to resolve them, in particular where they are your own and you are simultaneously releasing a set of projects.

Utilise Source Control

If you are using a source control system, the release should be cleanly built from a tag (so that the release codebase is clearly known). So, the SCM needs to be tagged before the release is cut. Maven can provide assistance with this.

Currently, the only automated release mechanism in Maven for cutting releases is provided by the SCM plugin. Please refer to the documentation on Making Releases from SCM.

The SCM plugin cuts a release in two steps:

1. preparation - where the descriptors are updated, committed and the code is tagged
2. performing - where the code is checked out from the SCM cleanly and built from scratch, then released using the distribution goals you selected in the first section.

Cutting the Release

As discussed previously, if you are using an SCM, you should utilise the SCM plugin to do this as this will ensure a build against a clean checkout.

Ideally you should also build in a known environment each time.

Pay attention to the version of Java used to compile the release: some bytecode produced by later versions may be incompatible with earlier versions. Look out for warnings provided by Maven about this. Maven can offer assistance: setting `maven.compile.target` of the Java Plugin passes the target JVM to the compiler. Note (however) that some more subtle compatibilty issues may be introduced by some modern compilers which can be fixed only by using an older compiler.

With everything together and prepared, cutting the release is quite simple (whether the project uses SCM or not). If the plugin supports it, you can deploy the released artifact directly to the repository.

While Maven will currently create and deploy an MD5 checksum along with your distribution, this is only useful for verifying download integrity when the sum can be download from a trusted repository. In many environments (including where there is public distribution over the internet) it is therefore recommended that additional measures be put in place to allow the authenticity of the release to be verified.

It is therefore recommended that (in these cases), an OpenPGP compatible digital signature is also created. The original signatures and (most importantly) the public keys used to sign the releases should be held on a tightly secured central server. (Most users will need to download the public key but may not be able to verify it using the web of trust. It is therefore vital that they can download the key from a secure server.) The public key should also be made available through a public key server (for example, the MIT key server) and efforts made to link it strongly into the web of trust. It is of crucial importance that all private keys used to sign releases be kept secure.

Since Maven does not currently support automatically generating digital signatures, it is recommended that you sign your release using GPG (or some other OpenPGP compatible application).

Republish the Site

As part of the deployment, you might want to republish your Maven generated web site with the new release information, download links and release notes.

Announce the Release

Finally, after the release is out and checked, you will want to make announcements to tell the world about it. Here again, Maven can help. (If you are using a sophisticated distribution system with mirroring, you should wait until the mirrors have sync'd before making the annoucements.)

If you are maintaining a change record, you can use the Announcement plugin to generate a plain text announcement "release notes" style announcement and have it mailed to an announcement or other mailing list.

2.2.7 Web Applications

Web Applications

Building a web application is not too much different to working with other types such as JARs in Maven. You still create a project descriptor, add your source code and resources, and build an archive to finish.

However, a few extra steps must be taken to create the appropriate web application structure.

Adding Webapp Contents

The first step is to add the web application contents. Unlike a JAR, resources are not included in the root of the WAR - they will be located in `/WEB-INF/classes`, as the purpose of the resources is to make them available in the classloader inside the application.

To include the contents of your web application (JSPs, images, etc., as well as some of the contents of WEB-INF), create a directory called `src/main/webapp`. These will all reside in the base of the web application when it is packaged up.

To configure the web application to use these resources, set the property:

```
maven.war.src=${basedir}/src/main/webapp
```

You can include whatever you like in this structure, including a `WEB-INF` directory and anything underneath it. However, you should not have a `/WEB-INF/classes` or `/WEB-INF/lib` directory as Maven will populate this from your project.

As mentioned, resources and classes will be compiled and copied into the `/WEB-INF/classes` directory. The `lib` directory is populated from the project's dependency list, as shown in the next section.

Adding Dependencies to the lib Directory

Since Maven requires that you declare all of the dependencies of your project, it is quite easy to also set up the `lib` directory in your web application.

Note however, libraries are not bundled into the web application by default - you must list those that you want included. This is quite simple to do, by adding a property to those dependencies that are required in the WAR:

```
<dependency>
  <groupId>mycompany</groupId>
  <artifactId>mycompany-utils</artifactId>
  <version>1.0.0</version>
  <properties>
    <war.bundle>true</war.bundle>
  </properties>
</dependency>
```

Building and Deploying

There are two particular goals that are used to build a web application: `war:webapp` and `war:war`.

The first builds an unpacked directory structure in the `target` directory that is the same as it would be exploded into an application server. This can be most effective for deploying directly into an application server with reloading enabled in the development environment so that reconstructing the WAR in this directory after making changes will be loaded.

The second will build a `.war` archive which is later deployed into the servlet container as is.

To automatically deploy into a servlet container will rely on scripting up the appropriate goals or locating an existing plugin for the container. There is currently one available for JBoss - see the JBoss Plugin. Tomcat also has several deployment Ant tasks which can easily be used from Maven.

Further Information

If you would like an example, please refer to The `simple-webapp` example project.

For more information on goals and configuration, see the WAR plugin reference.

Precompiling JSPs for Tomcat

How to precompile JSPs is a common question when developing Web Applications, as doing so can bring a performance boost to the first time deployment of your application in a production environment.

The following code snippet can be used in `maven.xml` if you are using a Tomcat 4.1 based servlet container. It should also serve as a starting point for implementing a similar task for other containers.

```
<preGoal name="war:webapp">
  <j:set var="precompileJsp" value="${precompile.jsp}"/>
  <j:if test="${precompileJsp == 'true'}">
    <attainGoal name="precompile-jsp"/>
  </j:if>
</preGoal>

<postGoal name="war:webapp">
  <j:set var="precompileJsp" value="${precompile.jsp}"/>
  <j:if test="${precompileJsp == 'true'}">
    <maven:set var="target" property="maven.war.webapp.dir"
plugin="maven-war-plugin" />
    <util:available file="${maven.build.dir}/web-fragment.xml">
      <util:loadText var="fragment" file="${maven.build.dir}/web-fragment.xml"/>
      <ant:replace file="${target}/WEB-INF/web.xml" token="&lt;!-- [INSERT FRAGMENT
HERE] --&gt;" value="${fragment}"/>
    </util:available>
  </j:if>
</postGoal>

<goal name="precompile-jsp" description="Precompile all JSPs into java classes, and
then into classes" prereqs="war:load,java:compile">
  <maven:set var="warSource" property="maven.war.src" plugin="maven-war-plugin" />
  <ant:mkdir dir="${maven.build.dir}/jspc"/>
```

```
  <ant:mkdir dir="${maven.build.dir}/jspc-processed"/>
  <ant:mkdir dir="${maven.build.dir}/jspc-classes"/>

  <j:set var="jspOutDir" value="${maven.build.dir}/jspc"/>
  <j:set var="jspClassesOutDir" value="${maven.build.dest}"/>
  <ant:path id="jspc.classpath">
    <ant:pathelement location="${tomcat.home}/common/lib/jasper-runtime.jar"/>
    <ant:pathelement location="${tomcat.home}/common/lib/jasper-compiler.jar"/>
    <ant:pathelement location="${tomcat.home}/common/lib/servlet.jar"/>
    <ant:path refid="maven.dependency.classpath"/>
    <ant:pathelement path="${maven.build.dest}"/>
  </ant:path>
  <ant:taskdef name="jasper2" classname="org.apache.jasper.JspC"
classpathref="jspc.classpath"/>
  <ant:jasper2
    webXmlFragment="${maven.build.dir}/web-fragment.xml"
    package="${pom.package}.jsp.${pom.artifactId}"
    outputDir="${jspOutDir}"
    srcdir="${warSource}"
    uriroot="${warSource}"
    uribase="/${pom.artifactId}"
    verbose="2"/>
  <ant:javac
    srcdir="${jspOutDir}"
    destdir="${jspClassesOutDir}"
    debug="${maven.compile.debug}"
    deprecation="${maven.compile.deprecation}"
    optimize="${maven.compile.optimize}"
    classpathref="jspc.classpath"/>
</goal>
```

To execute this code, set the `precompile.jsp` property to `true`. When building the web application, the JSPs will be compiled and added to the generated WAR file.

These servlets will not be utilised by Tomcat by default. To do so, you must include them in the `web.xml` file. The script above does this, as long as you include the following comment in your `web.xml` file in between the last `servlet` and first `servlet-mapping` declaration:

```
<!-- [INSERT FRAGMENT HERE] -->
```

2.2.8 Multiple Modules

Multiple Modules

Maven supports the notion of a product with multiple modules. While each `project.xml` is a discreet unit of work, they can be gathered together using the reactor to build simultaneously. The reactor was originally created to encourage the creation, or refactoring, of projects into smaller, discrete, more coherent units.

The reactor determines the correct build order from the dependencies stated by each project in their respective project descriptors, and will then execute a stated set of goals. It can be used for both building projects and other goals, such as site generation.

While the reactor requires you to write some Jelly script to interface with it, there is also a friendly wrapper plugin called `multiproject` that can allow you to control the setup with simple properties.

Using the Multiproject Plugin

Using the plugin is quite simple. The default set up assumes that you are in the top level directory of your build, and that and subdirectories that contain a `project.xml` file are subprojects to build.

To change this behaviour, set the `maven.multiproject.includes` and `maven.multiproject.excludes` properties appropriately, for example:

```
# Foo is broken, don't build it
maven.multiproject.excludes=foo/project.xml
```

All of the artifact based goals have a multiproject equivalent:

`multiproject:install`	Build and install all projects into the local repository
`multiproject:install-snapshot`	Build and install a timestamped build of all projects into the local repository
`multiproject:deploy`	Build and install all projects, deploying to the remote repository
`multiproject:deploy-snapshot`	Build and install a timestamped build of all projects, deploying to the remote repository

For the plugin to determine how to build each project, you must define a property for each project that requires it:

```
maven.multiproject.type=war
```

The default type is `jar`.

Note that the original purpose of the Multiproject plugin was to assist in aggregating the web site of subprojects, and so this comprises a large amount of the plugin's functionality. Simply by running:

```
maven multiproject:site
```

This goal will build the sites of all subprojects, including the top level that it was run from and will build an appropriate navigation to access each of the subprojects, and an overview listing the subprojects.

The behaviour of this goal and all others can be easily customised. For more information, see the Multiproject plugin reference.

Finally, you may wish to run some other goal(s) over the set of projects. To do this, run the following goal:

```
maven multiproject:goal -Dgoal=goal1,goal2
```

Using the Reactor

While the Multiproject plugin covers most common use cases in a convenient manner, in some instances it is necessary to access the reactor directly to execute multiple projects.

The normal use of the reactor tag is as follows:

```
<maven:reactor
  includes="**/project.xml"
  excludes=""
  banner="Building"
  goals="goal1, goal2"
  ignoreFailures="false"
/>
```

This will loop through every `project.xml` file found in the current tree, in the order defined by the dependencies they have on each other. The goals listed will be executed on each project in sequence.

The reactor tag is described in the Maven Jelly Tag Library reference.

Processing Projects after the Reactor

In some instances, you will want to work with the projects that the reactor did in your own script afterwards. To do this, you can utilise the `${reactorProjects}` variable set in the Jelly context after each reactor execution. This is a list, and so can be iterated. Each variable in the list will be a project object, and can be used in the same way as `${pom}` would normally be used.

If you'd like to gather a list of subprojects without executing any goals on them, this can be done by using the `collectOnly` attribute of the `maven:reactor` tag.

In some cases, you may wish to process a whole set of projects, and handle the failed projects differently. This requires that you set the `ignoreFailures` attribute to `true` in the `maven:reactor` tag. After all the projects have finished processing, the context variable `${failedProjects}` will be available to iterate through and projects that failed to build completely.

If `ignoreFailures` is not set, it defaults to `false` and the entire build will end when one project fails to build.

2.2.9 Handling Dependencies

Handling Dependencies

One benefit of Maven is that it can make you much more aware of dependencies your project has on other libraries. This outlines some ways to keep track of your dependencies, and to keep them in sync. This is particularly the case in a multiproject scenario.

Overriding Stated Dependencies

You may find it convenient, or necessary, at times to override the dependencies stated in a given POM. You may wish to use JAR artifacts somewhere in the filesystem or you may wish simply to override the stated version of a given JAR artifact. For this Maven provides an easy way for you to select which artifacts you want to use for building.

In order to use the JAR override feature, you must set the `maven.jar.override` property to `on`. Once this property is set you can specify JAR override directives in any of the properties files that Maven processes.

There are two type of JAR override directives. The first form allows you to specify a specific JAR artifact path for a given artifactId; the second allows you to specify a specific version of a JAR artifact that exists in your local repository. The two forms are as follows:

```
maven.jar.artifactId = [path]
maven.jar.artifactId = [version]
```

Below is an example of what a properties file might look like that contains JAR override directives:

```
maven.jar.override = on

# Jars set explicity by path.
maven.jar.a = ${basedir}/lib/a.jar
maven.jar.b = ${basedir}/lib/b.jar

# Jars set explicity by version.
maven.jar.classworlds = 1.0-beta-1
```

Version Consistency

The version override is of particular assistance when you are attempting to maintain a consistent version across different subprojects, and is a much better alternative to using entities or property interpolation.

The situation is that you have a number of subprojects using similar, but not identical dependencies - but that when a particular dependency is used you want to ensure the version is consistent.

You can use inheritence to set up the structure, but can't put the dependency in the parent POM as all subprojects will inherit that particular dependency whether they want it or not.

The best way to implement it is to put the dependency in each subproject as required, but put the *version override* in the parent POM. Wherever the dependency appears, that version will be used, but the dependency will not be used unless it is actually declared.

Using SNAPSHOT Dependencies

In Maven versions containing the keyword `SNAPSHOT` are considered special. They approximate the latest development build of a project that has been deployed to the repository. If a project that you depend on is changing frequently you can state in your POM that you wish to try and keep up with that project by declaring it a SNAPSHOT dependency. So, for example, you may be trying to stay abreast of changes in Jelly so you might put the following in your POM:

```
<dependency>
  <groupId>commons-jelly</groupId>
  <artifactId>commons-jelly</artifactId>
  <version>SNAPSHOT</version>
</dependency>
```

Assuming that project is publishing a version called `SNAPSHOT` (which happens when the `jar:deploy-snapshot` goal is called), then each time you build, Maven will check the remote repository for changes in that JAR and download it again if a newer `SNAPSHOT` is available.

Note: It seems to be a common misconception that Maven would replace `SNAPSHOT` by the most current version of a project that is available in the repository. This is not the case: using a `SNAPSHOT` dependency requires that the project that you depend on has actually published a `SNAPSHOT`, and only this `SNAPSHOT` will get updated automatically.

If you are working offline Maven will warn you that your SNAPSHOT dependencies may be out of date. This is just a warning, and as the remote copy often does not change frequently it can be ignored.

Note: the version need only contain the word `SNAPSHOT` - it does not need to equal it exactly. It is traditional to append it to a version to indicate "development towards version X". For example, `1.2-SNAPSHOT` is the development version between 1.1 and 1.2.

Plugin Dependencies

It is a little known feature that you can include a dependency on a specific version of a plugin in your `project.xml`, for example:

```
<dependency>
  <groupId>codeczar-tomcat</group>
  <artifactId>maven-tomcat-plugin</artifactId>
  <type>plugin</type>
  <version>1.1</version>
</dependency>
```

This causes the plugin to be installed into the local repo, and expanded into the cache, but it will not be installed into the maven plugins directory. In other words, your project will use that particular version of

the plugin on the fly, without messing up your Maven installation. You may also force other developers to use the same version of the plugin, even if they have a different version of the same plugin installed.

Keeping Track of Dependencies

Maven provides two site reports that can assist in keeping track of dependencies.

The standard dependency report included in the "Project Info" section of every site lists out the information for each dependency in the project, including those inherited. This is mostly useful to new users of your application or library that wish to see what additional dependencies they will require to use it.

The other report is given by the multiproject plugin, and is called the *Dependency Convergence Report*. This report will look for matching dependencies with different versions in your multiproject set up, so that you can synchronize those using different versions. The section above on Version Consistency can help automate this.

2.2.10 Best Practices

Best Practices

One of the biggest advantages of Maven is that it makes things easiest when best practices are followed. There are several recommended steps to take in setting up your build and developing the project that will help detailed here.

Follow Conventions

Many have complained that Maven forces a certain structure on a project. While this is not completely true (most settings are still configurable), it certainly does make it easier to use defaults.

From the layout of your source code to the layout of your project documentation, and all the standard goals in between, following conventions helps new users find their way around your project with minimal training and additional documentation.

For more information, refer to the Conventions Reference.

Reproduciblity

It is important to make sure a build is reproducible. This means a few things:

1. That any user can check it out and build as is (or with minimal configuration)
2. That you can check out a historical codebase and it should build as it did then
3. That a generated artifact for a particular version of a project should always be built identically

Making sure any user can check it out and build means that you must avoid the need for properties specific to your environment. The best practices to follow here are:

- Minimise the number of properties in `$HOME/build.properties` and ensure they are only those required for your particular environment, and that it is well known that they need to be set (eg HTTP proxy settings, J2EE server installation path)
- Don't use absolute paths in properties and scripts for a project
- If you must add an environment specific property, add it to `build.properties` in the project, but don't commit it to source control. Commit a sample properties file that other developers will know to copy and edit accordingly. However, wherever possible, use sensible defaults to avoid the need for this.

Being able to check out a historical codebase is more important in some environments than others, but often you don't know you need it until it is too late, so it is a good practice to follow.

- Follow the property rules above as they will also cause old builds to fail as things change in a local environment
- Make sure that releases use known versions of dependencies - don't depend on SNAPSHOTs that are subject to change and likely to be different the next time you build from when the original release way
- Document the version of Maven and plugins installed when building the project. This information

can be obtained with `maven --info`. Future versions of Maven aim to make this easier to keep consistent through the life of the project.

Finally, make sure generated artifacts are always built identically. This is important as two files with the same name, but different settings built in can become very confusing. Also, you often do not want to have to rebuild an artifact as it goes from staging to QA to production as it introduces the risk that it may be built differently (especially if the previous principles were not adhered to!) Some recommendations for this are:

- Avoid the need to filter resources. While this can be useful in a development environment, it usually requires rebuilding of an artifact between different phases of deployment. The best alternative is to externalise the configuration - for example in J2EE (where this is a common occurrence), make sure all configurable information such as database connection properties are in the deployment descriptor, provided through JNDI outside of the webapp or other deployable item. This means the particular artifact can be deployed identically into different servers, with just the external configuration differing.
- Only interpolate values in the POM with other POM values, not properties that might be user or environment specific.
- Avoid the use of file entities in the POM.
- Follow the other rules above - avoid including resources outside of the base directory of the project, and don't use properties customised to a particular environment.

One helpful method to assist reproducibility, if your environment has the capacity, is to set up a user on a machine with a clean environment that matches the target environment, with known properties, plugins and other software. This user can be used to build releases from a known baseline.

Scripting

Try to minimise the amount of scripting done. Those familiar with Ant will often put a lot of Ant tasks into `maven.xml`. *You should try not to have a `maven.xml` file.*

The reason for this is that this starts to make each build behave differently, losing one of the benefits of having a uniform environment. It also requires more maintenance and documentation.

Try to look for plugins that provide the functionality you are looking for, configure existing plugins, or even make small rearrangements to your project if you are just looking to reproduce existing functionality in a different way.

If you do have to do scripting, it is highly recommended to create a separate plugin project that provides these services so that it does not need to be duplicated across projects and the changes are isolated.

Things that are often placed in `maven.xml` are shortcut names for commonly used goal combinations, specification of the default goal, and pre/postGoal definitions to make small modifications to the behaviour of an existing goal - for example to add a new source root before compilation.

Writing Plugins

Writing plugins is the best way to extend Maven and continue to share it among other projects. However due to its architecture there are some things that are not recommended.

Firstly, try not to use `pre/postGoal` definitions inside a plugin. This will bind that plugin to another plugin causing side effects when the plugin is loaded, but different effects when it is not. A better solution is to define the behaviour in a new goal, and then have the projects using the plugin define the `pre/postGoal` to call the new goal.

Also, try not to call into other plugins using `attainGoal`. This makes your plugin susceptible to changes in the other plugin. Often you won't have an option - however the recommended way for plugins to expose their functionality is using Jelly Tag Libraries, or even better - shared Java code.

Use of `prereqs` is strongly preferred over `attainGoal` in any situation other than inside `pre/postGoal` definitions. This ensures the best order can be determined and that goals are only executed once.

Project Development Cycle

Whenever you start making changes after a recent release, you should check the `currentVersion` tag in `project.xml` and ensure that it is *`nextVersion`*`-SNAPSHOT`.

By tagging the version as `-SNAPSHOT` it signifies it is not released (so won't accidentally overwrite the official release), and has the advantage that when published, newer versions will be downloaded if it changes.

Part of the release process is to set `currentVersion` to the actual version released. For more information, set Making Releases.

Keep the Site Documentation Updated

The `xdocs` directory should contain a set of documentation that can be published as a site, giving users and developers instructions on how to use the project.

When you make changes to the project, make sure the documentation is kept up to date. For plugins, consider especially the `xdocs/goals.xml` and `xdocs/properties.xml` files. If the plugin has no xdocs, you can generate skeletons using `maven plugin:generate-docs`.

Maintain a Change Log

The `xdocs/changes.xml` file can be used to track changes as they are made, including who made them, what ticket number they relate to, and who they were contributed by if it was submitted as a patch.

Maintaining this gives a more readable change log than that provided by the SCM reports, and can be used to generate documentation and release notes.

When committing a change, you should edit and commit `xdocs/changes.xml` (create it if it does not exist) and describe the change according to the format given in the changes plugin.

Note: In the future, Maven should be able to integrate tightly with your issue tracking system to reduce the need for this.

Refactor Your Build

You will find that it is much easier to work with Maven when projects are simple and independent. This kind of loosely coupled structuring is often recommended for source code, so it makes sense to apply the same thought process to the structure of your build.

Don't be afraid to split an artifact in two if it is becoming complicated to maintain the build.

Some specific examples:

- If you have some sets of integration or functional tests that rely on the artifact being fully packaged and have a large amount of configuration and/or code of their own, consider putting them in a separate project that depends on the main artifact.
- If you need to share test code among different projects, put that test code in its own artifact (this time included in the main source tree) and depend on that artifact for your tests in other projects.
- If you are generating a large amount of code, you may find it helpful to keep the generated code in an artifact of its own, and depend on that from your other projects.

Getting ready for Maven 2

Dependency Scopes: Maven 2 has a concept of *scopes* which are used to control when and where the dependencies are used in terms of e.g. the compile, test and runtime class paths. Maven 1 doesn't have this concept but it's still useful to include this information when adding dependencies to your project. It will be useful when you (possibly) convert your project to Maven 2 and if your repository is ever converted to a Maven 2 repository the Maven 2 POMs will include dependencies with the correct scope.

To set the scope on your dependency use the *scope* tag like this:

```
<dependency>
  <groupId>foo</groupId>
  <artifactId>bar</artifactId>
  <version>1.0</version>
  <properties>
    <scope>test</scope>
  </properties>
</dependency>
```

2.2.11 IDE Integration

Using Maven in your IDE

There are three ways to use Maven with your IDE:

- Integrated into the project support
- As an external tool
- Using Maven to generate your IDE project settings

Support for IDE integration is provided by the MevenIDE project. Currently, they support project editing in NetBeans 4.0+ and Eclipse.

Using Maven as an external tool is usually quite simple, and allows you to run Maven goals from within your IDE. Specific instructions for setting up your IDE are listed here:

- IntelliJ IDEA

Maven also provides plugins for several IDEs that will generate project files from your Maven project descriptor. Please refer to the plugin documentation for your IDE:

- IntelliJ IDEA
- Eclipse

2.2.11.1 IntelliJ IDEA

Using Maven in IntelliJ IDEA

The recommended way to integrate Maven into your IntelliJ IDEA, is to use the MevenIDE for IDEA plugin. It will allow you to execute goals from within the IDE, visually edit the POM (project.xml) file, select favorite and common goals for each module, etc. The MevenIDE for IDEA project is still young, but provides most of the basic tools you will need to use Maven from IntelliJ.

Alternatively, you can configure it as an external tool, which allows you to define command lines that execute Maven with customizable parameters. To add Maven as an external tool in IntelliJ IDEA 4.0+, open the IDE settings, and click the "External Tools" button. From there, press "Add..." to open a new dialog. The following settings should be used:

Name:
Group:
Description:
Program:
Parameters:
Working Directory:

When selected from the tools menu, this will prompt you for the goals to run, and run it from the main project directory.

There are other tools you might like to set up in a similar fashion. You can quickly do this by pressing the "Copy..." button and renaming the new tool. Here are some examples:

- As above, but use `console` as the parameters instead of `$Prompt$`. This will start Maven's console in a small pane inside IntelliJ
- As above, but use `idea` (or `idea:multiproject`) as the parameters. Name this "Regenerate IntelliJ project". When run, it will regenerate all your IntelliJ projects, and IntelliJ will prompt you to reload your project. For more information, see the IntelliJ IDEA Plugin documentation.
- As above, but use `$FileDir$` instead of `$ProjectFileDir$`. This is useful in a multiproject setup to run a specific goal on a particular module. In this case you should select the "context menu" option instead of the "tools menu" option so that you can right-click on a file in the navigation pane.

Finally, you might like to set up an output filter. In the external tools dialog, press the "Output Filters..." button, and use the following settings:

Name:
Description:
Regular expression to match output:

2.3 Customising Maven

Customising Maven

Earlier in the User's Guide, you've seen how to configure a `project.xml` to build various different types of projects.

In addition to customising the build through the project model, additional properties can be set, and scripting can be used to define custom goals, or enhance the workflow during the build.

Extending the Build with Properties

There are two sets of properties: a small set of standard properties that are fundamental to the operation of Maven, and can be changed to customise for your particular environment or change the basic structure Maven uses for a project.

This properties reference above also explains the order in which properties files are loaded, and the reasons to use each.

The majority of properties in Maven are the plugin properties. These can be used to customise any aspect of the build process.

For example, if you are looking to change the way java sources are compiled, you can see the reference for the Java plugin properties. This allows you to set properties such as the `-target` setting, and whether to see deprecations. These can be added to `project.properties` for all builders of your project:

```
maven.compile.target=1.1
maven.compile.deprecation=on
```

While several of the most helpful properties are explained in the guides on this site, there is a full list of plugins available on the Plugins Reference mini-site. From there, you can navigate to the plugin you are interested in (which matches the first part of the goal name you are calling and wanting to configure), and on to its list of available properties.

Note that while properties are essential to configuring Maven and it's plugins, they can also be a source of bad practices. See the Best Practices guide for more information.

Note that due to a 'feature' in the jexl expression evaluator, property names (not values!) must not contain dashes (which are interpreted as subtraction operators by jexl).

Property Substitution in the Project Descriptor

It is often much easier to parameterise values in the project descriptor (or in the properties files) with other properties or fields from the project descriptor than to redefine them each time. This is particularly powerful when used with inheritence, as seen in the next section.

For example, you might set the URL based on the artifact ID:

```
<project>
  <artifactId>my-project</artifactId>
  ...
  <url>http://www.mycompany.com/mainproject/subprojects/${pom.artifactId}</url>
</project>
```

For more information on how to specify expressions to use for interpolating these values, see the Jexl section of the Scripting Reference.

Project Inheritence

When working with multiple projects that share some information, it would be quite tedious to have to enter the same information into each project, or to maintain it afterwards.

For this reason, it is possible to extend a common project file:

```
<project>
  <extend>../project.xml</extend>
  ...
</project>
```

All elements are inherited and merged in, with later definitions overriding any defined earlier. Projects can be inherited to any level, but each can only extend one parent.

If there are any properties to be substituted into the parent POM or properties, the value will be evaluated *after* inheritence has been completed. For instance, if in the parent POM with artifact ID `my-parent` there was a description set like so:

```
<description>This is ${pom.artifactId}</description>
```

In the child project, if description were inherited and the artifact ID were `my-child` then the description would be `This is my-child`.

In addition to the `project.xml` definition, properties files and `maven.xml` files associated with the parent POM are inherited. Like the project descriptor properties and goals defined are added to those defined in the subprojects, with the subproject properties and goals overriding those from the parent if specified. `preGoal` and `postGoal` definitions that are inherited are accumulated though - all specified hooks will be executed.

Note: lists in the POM will overwrite lists defined in parent projects, with the exception of dependencies where the lists are accumulated.

Scripting Maven

If the provided plugins do not give all the flexibility needed to do the required tasks, you can add scripting to your project to go the last steps.

There are two ways to add additional goals and hooks to your project - through `maven.xml`, and by writing your own plugin. It is generally recommended to write your own plugin for any significant scripting effort as it will make it reusable across projects, separated, and allows you to use Java code as well as basic Jelly script.

The `maven.xml` file is tied to the `project.xml` file in the same directory, in the same way as `project.properties` is. This means that it is loaded and its goals processed whenever your project is, and also means that it is inherited from any projects you extend.

The format of `maven.xml` is typically like this:

```
<project default="jar:jar" xmlns:j="jelly:core">
 ...
</project>
```

The `project` element is the root element, and can specify a default goal name for the project. The namespace definitions are used by Jelly to import tag libraries.

Within this file, you can define new goals, or attach hooks to existing ones, write logic using Jelly tag libraries, and use existing Ant tasks to add build behaviour.

For more information on scripting in general, please see the Scripting reference. For particular information on building plugins, see Developing Plugins in this User's Guide.

As with properties, there are also some best practices to adhere to with regards to scripting Maven to ensure the build is maintainable.

2.3.1 Scripting

Scripting References

Customised scripting in Maven is achieved in one of two ways:

- Creating a `maven.xml` file containing custom goals
- Creating your own plugin containing custom goals

Note that there are several best practices related to scripting that it is worth adhering to.

All scripts in Maven 1.x are written in the Jelly XML scripting language.

For an explanation of how to develop a plugin, please see Developing Plugins, or for details about `maven.xml` see the section on Customising Maven, both in the User's Guide.

Jelly

Maven uses Jelly as it's scripting language, and any valid jelly tags can be placed in the `maven.xml`, or inside a plugin.

Here's an example `maven.xml`:

```
<project xmlns:j="jelly:core" xmlns:ant="jelly:ant" xmlns:u="jelly:util">
  <goal name="nightly-build">
    <j:set var="goals" value="java:compile,test" />
    <ant:mkdir dir="${maven.build.dir}" />
    <u:tokenize var="goals" delim=",">${goals}</u:tokenize>
    <j:forEach items="${goals}" var="goal" indexVar="goalNumber">
      Now attaining goal number ${goalNumber}, which is ${goal}
      <attainGoal name="${goal}" />
    </j:forEach>
  </goal>
</project>
```

XML Namespaces are used to declare the Jelly tag libraries you want to use, much like JSTL (in fact, many of the Jelly tag libraries closely follow their JSTL equivalents).

More information on Jelly can be found at:

- The Jelly Project Home Page
- The Jelly Tag Libraries
- Dion Gillard's blog about Jelly examples:
  1. Core Jelly - structuring scripts using import
  2. The Jelly Util Taglib

Jexl

Jexl is the expression language used by Jelly. This allows you to get the values of properties, but also to

execute functions on the Java objects behinf those variables. For example:

- `${myVar}` gets the value of the variable `myVar`
- `${empty(myVar)}` calls the "empty" function in Jexl to test if the string is null or 0-length
- `${pom.inceptionYear.equals('2001')}` tests equality
- `${pom.build.resources.isEmpty()}` calls the `isEmpty()` method on the `List` given by `pom.build.resources`.

For more information about Jexl, see it's home page.

Ant

It is important to note that most of the functionality in Maven 1.x comes from the equivalent Ant tasks. To check the functionality of that task, or use it within your own script, refer to the Ant Documentation.

Using Werkz in Jelly

Werkz is the library used to handle the goal chain in Maven. You can use Jelly scripts to define new goals for works or attach actions to existing ones. This is described in the sections that follow.

Declaring Goals

Declaring new goals is quite simple and is done in both `maven.xml` and plugins. To declare a goal add code such as this:

```
<goal name="my-goal" prereqs="some-other-goal" description="My Goal">
  ...
</goal>
```

The goal name can be anything you desire. If an existing goal exists by that name is defined, any given in `maven.xml` takes precedence (overriding plugins). If it exists in multiple plugins, then the results are undefined. For this reason, goal names in plugins are usually prefixed with the plugin name, eg: `java:compile`.

`prereqs` is a comma separated list of goals that must be executed before this goal is. If they have already been executed, they will not be run again.

The description is used for generating plugin documentation, and for displaying on the command line when the `-g`, or `-P` parameters are used.

Calling Goals

If, within a goal, you need to execute another goal, it can be done using:

```
<attainGoal name="my-goal" />
```

This is usually used in `preGoal` and `postGoal` definitions that can not specify `prereqs`, as you'll see in the next section.

Customising Goals

Customising existing Maven goals is quite simple as long as you know where to hook in to. As an example, let's consider you want to generate some resources and have them placed into the produced JAR file.

Note: Generally resources should be specified in the `<resources>` section of the `project.xml` file. However, if they are generated as part of the build, you will want to copy them to the destination yourself.

So, here is a sample `maven.xml` file that demonstrates how this would be achieved:

```
<project xmlns:ant="jelly:ant">
  <!-- The java:jar-resources goal copies resources into target/classes
       so that they are included in the JAR at / -->
  <postGoal name="java:jar-resources">
    <attainGoal name="generate-dynamic-resources" />
    <!-- maven.build.dest is the location where resources are put -->
    <ant:copy todir="${maven.build.dest}">
      <ant:fileset dir="${maven.build.dir}/generated-resources" />
    </ant:copy>
  </postGoal>

  <!-- Here is the goal you've defined to generate the resources in some fashion -->
  <goal name="generate-dynamic-resources">
    <!-- Place them into ${maven.build.dir}/generated-resources in accordance with
         the fragment above -->
    ...
  </goal>
</project>
```

Accessing the Current Project

The current project is always available under the context variable `${pom}`. This allows the retrieval of project variables in your scripts. This is the same as how they are used within the project file itself when it is interpolated.

The variables and functions you can access are those listed in the javadoc. For example:
`${pom.artifactId}` calls `getArtifactId()` to retrieve the artifact ID.
`${pom.getVersionById(pom.currentVersion).tag}` gets the tag associated with the version which has an ID equivalent to the current version of the project.

Note that inside a plugin, the `${pom}` variable refers the project currently being built, not the project that the plugin is defined in. To access the project variables of the plugin itself, use the variable `${plugin}`.

Using Dependencies

There are three ways to access the dependencies declared on the current project being built.

The first is to use the entire set as an Ant path reference, if you are only passing it to an Ant task. The reference that contains all the dependencies in the classpath is `maven.dependency.classpath`. For example:

```
<javac ...>
  <classpath refid="maven.dependency.classpath" />
</javac>
```

If you need the path to a specific dependency, you can use the `getDependencyPath()` method. For example:

```
<ant:path id="my-classpath">
  <ant:path refid="maven.dependency.classpath"/>
  <ant:pathelement path="${maven.build.dest}"/>
  <ant:pathelement location="${plugin.getDependencyPath('junit:junit')}"/>
</ant:path>
```

Finally, you can iterate through the artifacts within a project to process them individually. For example:

```
<j:forEach var="lib" items="${pom.artifacts}">
  <j:set var="dep" value="${lib.dependency}"/>
  <j:if test="${dep.getProperty('war.bundle')=='true'}">
    <ant:copy todir="${webapp.build.lib}" file="${lib.path}"/>
  </j:if>
</j:forEach>
```

Working with Plugin Properties

As you will have seen many plugins have their own properties. You can set these values in the current project before the plugin is initialised, and they will be recognised. However, once it is initialised, setting them in the current project will have no effect, and changes made within the plugin are not visible to the current project.

If you need to get a property from the plugin, you can use the `maven:get` tag. For example:

```
<maven:get var="out_var" plugin="maven-war-plugin" property="maven.war.final.name"
/>
```

Note: you may also see references to a `maven:pluginVar` tag, and the expression `${pom.getPluginContext('maven-war-plugin').getVariable('maven.war.final.name')}`. These are both deprecated forms of the above tag.

Likewise, you can set a plugin property using the `maven:set` tag. For example:

```
<maven:set plugin="maven-war-plugin" property="maven.war.final.name"
value="${in_var}" />
```

2.3.2 Writing a Plugin

Developing Plugins

Adding functionality to Maven is done through the Maven plugin mechanism. Maven comes shipped with numerous plugins, and there are several 3rd party plugins. Plugins can be used to do virtually any task desired. For example, they can generate reports, create and deploy software distributions, run unit tests, and much more. This section of the document describes how to write your own plugin.

A plugin can contain the following:

- A `project.xml` file to describe the plugin project (required);
- A `plugin.jelly` file containing Jelly script for the plugin (required);
- A `plugin.properties` file to define any property defaults (optional);
- Plugin resources for adding to the classpath when the plugin is run (optional);
- Java sources, which are accessed from the Jelly script (optional).

How to create your first plugin

To start creating a plugin, you must first create a Maven project. This is the same as creating any other project, for example one that builds a JAR.

In a new directory, create a `project.xml` file like so:

```
<project>
  <pomVersion>3</pomVersion>
  <artifactId>hello-maven-plugin</artifactId>
  <groupId>hello</groupId>
  <name>Maven Hello World Plugin</name>
  <currentVersion>1.0</currentVersion>

  <!--
    You might want to include additional information here
    eg the developers, organisation, and dependencies
  -->

  <build>
    <!-- This is only required if you have Java code -->
    <sourceDirectory>src/main/java</sourceDirectory>
    <!-- These are only required if you have unit tests for your Java code -->
    <unitTestSourceDirectory>src/test/java</unitTestSourceDirectory>
    <unitTest>
      <includes>
        <include>**/*Test.java</include>
      </includes>
    </unitTest>

    <!-- This section is compulsory -->
    <resources>
      <resource>
        <directory>${basedir}/src/plugin-resources</directory>
        <targetPath>plugin-resources</targetPath>
      </resource>
```

```
      <resource>
        <directory>${basedir}</directory>
        <includes>
          <include>plugin.jelly</include>
          <include>plugin.properties</include>
          <include>project.properties</include>
          <include>project.xml</include>
        </includes>
      </resource>
    </resources>
  </build>
</project>
```

With just that, you can actually create the plugin (though it will do nothing!)

```
maven plugin
```

This command will create a JAR `target/hello-maven-plugin-1.0.jar`, but it is not installed yet. First, let's add some functionality.

Currently, all plugins must have a `plugin.jelly` file. This is a Jelly script, and looks identical to a `maven.xml` file in an individual project. Here is a script with a single, simple goal.

```
<project xmlns:ant="jelly:ant">
  <goal name="hello:hello" description="Say Hello">
    <ant:echo>Hello from a Maven Plug-in</ant:echo>
  </goal>
</project>
```

(For an explanation of what this means, please refer to Customising Maven, or the Scripting Reference).

Now that you can run the following command to install this into Maven's home directory:

```
maven plugin:install
```

You can prove the goal exists by running:

```
maven -P hello
```

Finally, run the goal from the plugin:

```
maven hello:hello
```

As mentioned, you can do anything in this script that you can do in `maven.xml`. For more information, please refer to Scripting in Maven.

Using Plugin Properties

While you can always reference the POM information in a Jelly script using `${pom.artifactId}`, for example, a plugin will often need to create new properties so that it can be customised.

The creation of these properties simply involves creating a `plugin.properties` file. **Note:** While plugins can also have a `project.properties` file, these do not specify properties exposed by the plugin. Instead, this is used to control building the plugin itself, like for any other project.

As an example, create a `plugin.properties` file in the same directory as `plugin.jelly`:

```
hello.greeting=Hi
```

Edit the `plugin.jelly` file so that the `<ant:echo>` line reads:

```
<ant:echo>${hello.greeting} from a Maven Plug-in</ant:echo>
```

Now, install the new plugin and run it (**note**, you do not need to run `maven plugin` first):

```
maven plugin:install
maven hello:hello
```

Notice the the new greeting. Now, to customise it:

```
maven -Dhello.greeting=Goodbye hello:hello
```

This property will be read from a variety of different places: `project.properties` for the current project, `build.properties` from your home directory, or a system property as above. The order of precedence is given in the Properties Reference.

Using Plugin Resources

You can use plugin resources to be able to load external files from inside a plugin. Let's say you wanted to copy an image into every site. First, add a file called `src/plugin-resources/hello-report.xml` to the source tree that looks like this:

```
<document>
  <body>
    <section name="Hello">
      <p>Hi!</p>
    </section>
  </body>
</document>
```

Next, you can create a new goal called `hello:report` in `plugin.jelly`.

```
...
  <goal name="hello:report">
    <ant:copy todir="${maven.gen.docs}" file="${plugin.resources}/hello-report.xml"
/>
  </goal>
...
```

The plugin resources become most useful when you have templates for generate site reports based on the project, which is discussed in the next section. This report goal will be used to "generate" the Hello Report.

Plugin dependencies

In maven 1.x, there are only two classloaders :

- root : Where are loaded ant, commons-logging, log4j.
- root.maven (child of root) : Where are loaded (by default) all others dependencies needed by maven, its plugins and the user project.

This design can create 2 problems when you write your own plugin :

1. You won't be able to use your own version of a dependency. If this one is already loaded by maven or one of its plugins in a different version you'll automatically use it. To avoid having this sort of problem, you are invited to use in priority the libraries versions already used by maven or by the bundled plugins.
2. If you try to use an Ant task which requires another dependency, the ant task doesn't find it even though you defined it correctly in your plugin's pom. This due to the fact that ant is loaded in the classloader "root" and your dependencies are visible in the classloader "root.maven". To solves this, you must specify to maven to load this dependency in the root classloader with :

```
<dependency>
  <properties>
    <classloader>root</classloader>
  </properties>
</dependency>
```

Providing Reports in a Plugin

Plugins are often used to provide one or more reports using a certain tool based on the information in a project. The standard reports are shown under "Project Reports" in the web site of any Maven project that has not disabled them. For example, the Unit Test Report shows which unit tests have passed and failed.

While any goal can perform a task, and transform the results into the web site, there are special steps to take to signify a report can be generated so that it can be listed on the web site, and enabled via the `<reports>` section in the POM.

For information on how to actually generate the report, it is easiest to refer to the Jelly source code of a

report that is similar to what you want. The end result should be to create an XDoc file in the directory specified by `${maven.gen.docs}`.

To register a report withing your project, you should define three goals (using hello as the example):

- `hello:report` - generates the report content. This was the goal we created in the previous section.
- `hello:register` - registers the report with Maven for inclusion on the site
- `hello:deregister` - disables the report so that a project can override a previous enabling command

Registering a report is done as follows:

```
<goal name="hello:register">
  <doc:registerReport
    pluginName="hello"
    name="Hello"
    link="hello-report"
    description="Greeting report." />
</goal>
```

This introduces the `doc` tag library, so the following change must be made to the project declaration:

```
<project xmlns:ant="jelly:ant" xmlns:doc="doc">
```

The parameters to the `registerReport` tag are:

pluginName	This is the start of the goal name to run, and represents this plugin. The report goal run will be `${pluginName}:report`.
name	The name of the report, which will be used to list in the report in the navigation bar of the web site.
link	This is a relative link to the final output of the plugin, with the extension omitted. When generating, the link should be relative to `maven.gen.docs`, so that it will end up as relative to `maven.docs.dest`. eg `${maven.gen.docs}/hello-report.xml` and `${maven.docs.dest}/hello-report.html`.
description	This should be a one line description of the output produced by your plugin. It is included on the auto-generated `maven-reports.xml` overview document.
target	This should be a target for the html link generated. This parameter is often used to open the report page in a new window of the browser (`target="_blank"`).

A plugin may specify more than one report. It is entirely possible to register additional reports as shown below (taken from the JavaDoc plugin included with Maven):

```
<goal name="maven-javadoc-plugin:register">
  <j:if test="${sourcesPresent == 'true'}">
    <doc:registerReport
      name="JavaDocs"
      link="apidocs/index"
      target="_blank"
      description="JavaDoc API documentation."/>
    <doc:registerReport
      name="JavaDoc Report"
      link="javadoc"
```

```
      description="Report on the generation of JavaDoc."/>
  </j:if>
</goal>
```

Notice that the above plugin does not register its reports if the project does not have any sources. It is encouraged that plugin developers use conditional logic to prevent their reports from registering if they don't apply to the user's project.

The final goal is the deregistration goal. This goal is optional, but should be included to allow users to force its removal from their site.

The `name` parameter should correspond to the original registration `name` parameter.

```
<goal name="hello:deregister">
  <doc:deregisterReport name="Hello"/>
</goal>
```

The final step is to add the report to your project to use it, using what was specified as `pluginName` when registering the report:

```
<reports>
  <report>hello</report>
</reports>
```

You can add this to any project, but for now we'll just reuse the same plugin project to test the plugin on. Add the above section to the `project.xml` you've created, then run:

```
maven site
```

Notice the final result of a `target/docs/hello-report.html` file. You can view this in your browser. When doing so, you'll also see the "Hello" item under the "Project Reports" menu in the navigation.

Sharing Your Plugin

Please see Sharing Your Plugin for guidelines on how to share your Maven plugins with the community.

Getting More Information

A good source of information is the source code for existing plugins. You can find the expanded plugin sources in `~/.maven/cache`.

For more information on scripting, refer to the Scripting Reference.

2.3.3 Sharing Plugins

Sharing Plugins

You've just created a killer Maven plugin. Now what?!

Where should plugins live? Who can update them? Does it really make a difference?

Can you put my plugin in the Maven distribution?

The main Maven distribution includes many plugins, but the growing trend is for plugin owners to host the plugins themselves (What do you mean, host?). Why? There are many advantages for the plugin owners, the users, and the Maven team.

Plugin owners are the best maintainers

Since only Maven committers may update the `maven-plugins` repository, changes you make to your plugin will take a while to make it into a Maven distribution. Each plugin needs a committer willing to maintain it, and that plugin is not the only piece of Maven the committer works on.

Plugin owners are in a much better position to maintain the plugin. They originally wrote the plugin. They have a vested interest in seeing it succeed. If they host the plugin themselves, then updates can be incorporated quickly, and users get the benefits of updated software quicker.

This benefit is often maximized when a team that writes a library, like the StatCvs group, also creates the Maven plugin. As the library changes, the plugin may be updated to keep in sync. In fact, the Cactus and StatCvs plugins were among the first to voluntarily remove their plugins from the Maven distribution and be hosted with their respective libraries.

Publishing Your Plugin

Step 1. How do I host my Maven plugin?

To "host" a Maven plugin, you require only two things:

- A place to keep your source code
- A website to deploy the following
  - Your plugin's site (`maven site`) documentation
  - Your plugin's JAR bundles for the repository upload requests

There are a variety of options available for each need. Your source code may simply be kept on your hard drive, using a local installation of a version control tool like CVS or Subversion (you do use version control, don't you?). If you'd like to share your plugin source with others, you can request a new project at a service like SourceForge or Java.Net, or request to join an existing project like the Maven-Plugins project on SourceForge.

If you're creating a plugin for an existing open-source (or even closed-source) library, the ideal situation is

to approach the library maintainers about hosting your plugin source and site. That way the plugin source is close to the people most likely to care about its success, and ensure it works well far into the future. The plugin site is located close to the already well-known library site. In the end, this cooperation means a better chance at happy users, and up-to-date plugins for everyone.

Step 2. OK. My plugin has a host, how can I put it on Maven's central repo?

If the project is part of the repository mirroring program, such as `maven-plugins.sourceforge.net`, publishing it at SourceForge will automatically mirror it to Maven's central repo without further work.

However, if the project is an independant, please see the instructions for uploading a resource to Maven's central repo.

NOTE: If your plugin requires resources (JAR files) that are not already in the repository, you may need to make several separate upload requests.

Step 3. My upload request was granted, but no one knows my plugin exists!

There are two ways to let your potential users know about your plugin. The first is through sending an announcement to the `maven-user` mailing list. Here's a few pointers on how to go about it:

- Review an excellent announcement sent by the StatCvs group, which includes plugin installation instructions.
- Combine the advice of the two parts to make your own effective announcement. You may like to use the Announcement Plugin to generate and perhaps mail the announcement.
- Make sure you've subscribed to the `maven-user` mailing list.
- Send your annoucement to the `maven-user` mailing list.

The second way to publicize your plugin is on the Maven site itself. The Maven project currently maintains a list of other sites that host plugins. Simply raise a request in JIRA to have a link added. Be sure to include the relevant information, following an existing plugin's description as an example.

Step 4. So now that they found my plugin, how do users install it?

Here's the general idea:

```
maven plugin:download -DartifactId=[artifactId] -DgroupId=[groupId]
-Dversion=[version]
```

For example:

```
maven plugin:download -DartifactId=maven-statcvs-plugin -DgroupId=statcvs
-Dversion=2.4
```

2.4 Repositories

Artifact Repositories

A repository (definition) 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 truely remote repository set up by a third party to provide their artifacts for downloading (for example, 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 it's central repository.

To override this, you need to set the property `maven.repo.remote` as follows:

```
maven.repo.remote=http://anothermavenrepository,http://repo1.maven.org/maven/
```

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

The protocols that are currently supported for a remote repository are `http`, `https`, `sftp` and `file`. Note that `sftp` is only supported since Maven 1.1-beta-3, and you'll have to set the username and password in the url, eg:

```
maven.repo.remote=sftp://${myusername}:${mypassword}@localhost/tmp/ble
```

(Future versions should allow you to use a setting of `${maven.repo.remote},http://anothermavenrepository`, but currently this causes an infinite recursion).

For more information on dependencies, see Handling Dependencies.

Building Offline

If you find you need to build your projects offline you can either use the offline switch on the CLI:

```
maven -o jar:jar
```

This is equivalent to:

```
maven -Dmaven.mode.online=false jar:jar
```

Or you can set `maven.mode.online` to `false` in your `build.properties` file to ensure you always work offline.

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.

If you would like normal operations to succeed, but for no dependency downloading to occur, set the `maven.repo.remote.enabled` property to `false`.

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 Uploading to Maven's central repository.

Usually, you will only be attempting to upload a release of your own application to an internal repository. See internal repositories for more information.

2.4.1 Internal Repositories

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.

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 Ibiblio.

It is **not** recommended that you scrape or rsync:// a full copy of Ibiblio as there is a large amount of data there. You can use a program such as Maven Proxy, running on 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 Standard Sun JAR Names 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 set the `maven.repo.remote` property in your project's properties:

```
maven.repo.remote=http://repository.company.com/
  or
maven.repo.remote=file:///shared/repository
```

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, or the filesystem. For example, to set up an SCP transfer, you define the following properties:

```
maven.repo.list=myrepo
maven.repo.myrepo=scp://repository.mycompany.com/
maven.repo.myrepo.directory=/www/repository.mycompany.com/
maven.repo.myrepo.username=${user.name}
maven.repo.myrepo.privatekey=${user.home}/.ssh/id_dsa
```

With this configured, the `*:deploy` goals will now send the published artifacts to the "remote" repository for sharing. This includes `jar:deploy`, `dist:deploy`, `war:deploy` and so on.

For more information, please refer to the Artifact Plugin Reference and the section on Making Releases.

2.4.2 Sun JAR Names

Standard Sun JAR Names

A common problem when trying to use Maven to build a project is if that project depends on a particular JAR from Sun that cannot be distributed via Ibiblio due to its license. (refer to this document for a history)

This particularly affects J2EE technologies, but includes some other reference implementations. Projects are now starting to develop clean room implementations of some of these specifications under open source licenses, however in many cases they are not complete, or projects are not using them.

The only alternative is to download the JARs yourself (accepting Sun's license), and to place them into your local or internal repository.

Should you need to use them, this document aims to list a standard location for the JARs so that all projects can reference them consistently, and so will only need to be downloaded and put in place once. It may not be complete, so if you have use for a Sun JAR that is not listed here, please contact the Developer's Mailing List so that it can be added.

The current list of JARs have been built up by convention. The rule is use the two first package levels as groupId and the name of the reference implementation jar as artifactId

Product artifact	Group ID	Artifact ID
Java Activation Framework	javax.activation	activation
J2EE	javax.j2ee	j2ee
JDO	javax.jdo	jdo
JMS	javax.jms	jms
JavaMail	javax.mail	mail
EJB 3	javax.persistence	ejb
J2EE Connector Architecture	javax.resource	connector-api
Java Authorization Contract for Containers	javax.security	jacc
Servlet	javax.servlet	servlet-api
Servlet JSP	javax.servlet	jsp-api
Servlet JSTL	javax.servlet	jstl
JDBC 2.0 Optional Package	javax.sql	jdbc-stdext
Java Transaction API (JTA)	javax.transaction	jta
Java XML RPC	javax.xml	jaxrpc
Portlet	javax.portlet	portlet-api
JNDI	javax.naming	jndi

2.4.3 Uploading to Ibiblio

Uploading to Ibiblio

So you want a resource uploaded to Maven's central repo?

To save time for everybody, here is how you go about it.

Step 1. Create an upload bundle

Use the artifact plugin, provided with the standard Maven distribution, to create an upload bundle:

```
maven artifact:create-upload-bundle
```

The bundle will be created in your "target" directory of the form:
`${pom.artifactId}-${pom.currentVersion}-bundle.jar`

If you are not using maven as your build system but want something uploaded to central repo, then you just need to make a JAR (using the `jar` executable, not `zip`, `pkzip` or equivalent) with the following format:

```
LICENSE.txt
project.xml
foo-1.0.jar (or whatever artifact is referred to in the project.xml)
```

Note that the bundle will be read by a script, so it must follow the above format. Also, the `project.xml` should at least contain the following elements:

- groupId (all lowercase)
- artifactId (all lowercase)
- name
- currentVersion
- dependencies

Some considerations about the **groupId**: it will identify your project uniquely across all projects, so we need to enforce a naming schema. For projects with artifacts already uploaded to central repo it can be equal to the previous used, but for new projects 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. More information about package names.

Examples:

- www.springframework.org -> org.springframework
- oness.sf.net -> net.sf.oness

Maven 2 projects

You can request a upload for a maven 2 project just using the maven 2 pom.xml instead of the maven 1

project.xml referenced previously. The jar will be uploaded to both maven 1 and maven 2 repos. Check this mini guide for more information.

Step 2. Posting the request

Post your request to JIRA. In the description you should write the URL of the upload bundle (if you're uploading more than one bundle please add all the urls under the same issue), then leave a blank line and provide the following:

- a url where the project can be found.
- if you are one of its developers, a url where your name or email can be found inside the project site.

This will speed up the uploading process.
You can place any additional comments you wish in the following paragraph. So the description field might look like:

```
http://wiggle.sourceforge.net/downloads/wiggle-1.0-bundle.jar

http://wiggle.sourceforge.net
http://wiggle.sourceforge.net/team-list.html

Wiggle is a fantastic new piece of software for automating the
clipping of nose hairs. Please upload!
```

Explanation

Some folks have asked why do we require the POM and license each time an artifact is deployed so here's a small explanation. The POM being deployed with the artifact is part of the process to make transitive dependencies a reality in Maven. The logic for getting transitive dependencies working is really not that hard, the problem is getting the data. So we have changed the process of uploading artifacts to include the POM in an attempt to get transitive dependencies working as quickly as possible. The other applications that may be possible having all the POMs available for artifacts are vast, so by placing them into the repository as part of the process we open up the doors to new ideas that involve unified access to project POMs.

We also ask for a license now because it is possible that your project's license may change in the course of its life time and we are trying create tools to help normal people sort out licensing issues. For example, knowing all the licenses for a particular graph of artifacts we could have some strategies that would identify potential licensing problems.

Maven partners

The following sites sync automatically their project repository with the central one. If you want a project from any of this sites to be uploaded to central repo, you'll have to contact the project maintainers.

- The Apache Software Foundation
- Codehaus
- MortBay Jetty

- OpenSymphony
- OS Java

2.5 Reference

Reference

The reference section contains the following documents:

Glossary
Conventions
Project Descriptor
Properties Reference
Maven Jelly Tag Lirary
Command Line Reference
IDE Integration

2.5.1 Glossary

Glossary

This document describes some of the most common terms encountered while using Maven. These terms, that have an explicit meaning for Maven, can sometimes be confusing for newcomers.

Term	Description
Project	Maven thinks in terms of projects. Everything that you will build are projects. Those projects follow a well defined "Project Object Model". Projects can depend on other projects, in which case the latter are called "dependencies". A project may consistent of several subprojects, however these subprojects are still treated equally as projects.
Project Object Model (POM)	The Project Object Model, almost always referred as the POM for brevity, is the metadata that Maven needs to work with your project. Its name is "project.xml" and it is located in the root directory of each project. To learn how to build the POM for your project, please read about the project descriptor.
Artifact	An artifact is something that is either produced or used by a project. Examples of artifacts produced by Maven for a project include: JARs, source and binary distributions, WARs. Each artifact is uniquely identified by a group ID and an artifact ID which is unique within a group.
Group ID	A group ID is a universally unique identifier for a project. While this is often just the project name (eg. `commons-collections`), it is helpful to use a fully-qualified package name to distinguish it from other projects with a similar name (eg. `org.apache.maven`).
Dependency	A typical Java project relies on libraries to build and/or run. Those are called "dependencies" inside Maven. Those dependencies are usually other projects' JAR artifacts, but are referenced by the POM that describes them.
Plugin	Maven is organized in plugins. Every piece of functionality in Maven is provided by a plugin. Plugins provide goals and use the metadata found in the POM to perform their task. Examples of plugins are: jar, eclipse, war. Plugins are written in Jelly and can be added, removed and edited at runtime.
Goal	Goals are what are executed to perform an action on the project. For example, the `jar:jar` will compile the current project and produce a JAR. Each goal exists in a plugin (except for those that you define yourself), and the goal name usually reflects the plugin (eg. `java:compile` comes from the `java` plugin).
Repository	A repository is a structured storage of project artifacts. Those artifacts are organized under the following structure: `$MAVEN_REPO/groupId/artifact type/project-version.extension` For instance, a Maven JAR artifact will be stored in a repository under `/repository/maven/jars/maven-1.0-beta-8.jar`. There are different repositories that Maven uses. The "remote repositories" are a list of repositories to download from. This might include an internet repository, its mirrors, and a private company repository. The "central repository" is the one to upload generated artifacts to (for developers of a company for instance). The "local repository" is the one that you will have on your computer. Artifacts are downloaded just once (unless they are a SNAPSHOT) from the remote repository to your local repository.
Snapshots	Projects can (and should) have a special version including `SNAPSHOT` to indicate that they are a "work in progress", and are not yet released. When a snapshot dependency is encountered, it is always looked for in all remote repositories, and downloaded again if newer than the local copy. The version can either be the string `SNAPSHOT` itself, indicating "the very latest" development version, or something like `1.1-SNAPSHOT`, indicating development that will be released as 1.1 (i.e. newer than 1.0, but not yet 1.1).
XDoc	XDoc is the format of documentation that Maven currently understands. It is quite simple, and allows embedding XHTML within a simple layout that is transformed into a uniform site. For information on how to create XDoc files, refer to the Building a Project Web Site document.

2.5.2 Conventions

Maven Conventions

This document defines some conventions that Maven recommends projects adopt. This is especially important if you intend to distribute your project publicly.

Artifact Naming

This section outlines the naming conventions used in the Maven project object model (POM). This document is an attempt to try and unify the many various ways projects name the artifacts that they publish for general consumption by the Java developer community (regardless of whether they are using Maven).

The first thing you will do when creating a project is to select a group ID and an artifact ID. If you are building a project to be part of a larger product that is already using Maven, you should attempt to follow any patterns already established by other projects for consistency.

These identifiers should be comprised of *lowercase* letters, digits, and hyphens only.

In general you should select a group ID that describes the entire product, and artifact IDs that are the basis of filenames for each item you distribute. The artifact ID may or may not overlap the group ID.

For example:

```
maven : maven-core
maven : wagon-api
```

As previously mentioned, the artifact ID should be the basis of the filename for the project, as by default Maven will use that and the version to assemble the filename. Having the version as part of the filename is strongly recommended to ensure that the version can be determined at a glance without having to check a possibly non-existant manifest, or compare file sizes with the official releases.

Following these guidelines are particularly encouraged when distributing via the Maven Repository, to ensure that it can easily fit alongside other projects and reduce the risk of conflicts and confusion.

Directory Structure

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 two sections document 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/
|  |  |  +- ...
|  |  +- resources/
|  |     +- ...
|  +- test/
|  |  +- java/
|  |  |  +- ...
|  |  +- resources/
|  |     +- ...
|  +- site/
|     +- xdoc/
|        +- ...
+- target/
|  +- ...
+- project.xml
+- README.txt
+- LICENSE.txt
```

At the top level files descriptive of the project: a `project.xml` file (and any properties, `maven.xml` or `build.xml` if using Ant). In addition, there are textual documents meant for the user to be able to read immediately on receiving the source: `README.txt`, `LICENSE.txt`, `BUILDING.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` or `.svn`, and any subprojects in a multiproject build (each of which would be laid out as above).

Note: currently, Maven 1.x violates this principle by defaulting the `xdocs` location to the top level directory. It is proposed that in future this be moved to `src/site/xdoc`.

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.

2.5.3 Project Descriptor

POM

This page is deprecated. The POM description is now available here :

http://maven.apache.org/maven-1.x/reference/maven-model/3.0.2/maven.html

2.5.4 Properties Reference

Properties Reference

In Maven, properties are used to customise the behaviour of Maven and its plugins.

Property Processing

Maven supports a hierarchy of different properties to allow specifying defaults and overriding them at the appropriate level.

The properties files in Maven are processed in the following order:

1. Built-in properties are processed
2. `${basedir}/project.properties` (`basedir` is replaced by the directory where the `project.xml` file in use resides)
3. `${basedir}/build.properties`
4. `${user.home}/build.properties`
5. System properties

The built-in properties are specified in the `plugin.properties` file of a plugin, or in `defaults.properties` within Maven itself.

Both the `project.properties` and `build.properties` files in the project directory are also inherited along with the related `project.xml` file.

The last definition takes precedence, so `${user.home}/build.properties` will override anything specified in a project, and properties given on the command line using `-D` will override everything.

Note: there are no per-user defaults, as there has not been a property shown where this concept makes sense. Currently, there are also no site-wide defaults, however this is planned for future versions of Maven.

The following table explains how each file should be used.

`${basedir}/project.properties`	These are properties specific to the project and can be used to set the values for plugin and Maven properties that are appropriate for that project. This file should be checked into your source repository and distributed.
`${basedir}/build.properties`	These are properties specific to the project, but also the user running it. It is for overriding values in the `project.properties` for this user only. It should *not* be checked into the source repository. If a user is required to set any of these properties for a build to run properly, a `build.properties.sample` file should be created and checked into the source repository as a courtesy.
`${user.home}/build.properties`	These are properties specific to the user but shared among all projects. This typically specifies remote repositories to download from and upload to, proxy settings, file system locations, and so on.

Built-in Maven Properties

The following properties are built-in to Maven and apply to all projects.

Maven Configuration Properties

Property	Description	Default Value
`maven.home.local`	The directory on the local machine Maven uses to write user specific details to, such as expanded plugins and cache data.	`${user.home}/.maven`
`maven.plugin.dir`	Where Maven can find it's plugins.	`${maven.home}/plugins`
`maven.plugin.user.dir`	Where Maven can find plugins for this user only.	`${maven.home.local}/plugins`
`maven.plugin.unpacked.dir`	Where Maven expands installed plugins for processing.	`${maven.home.local}/cache`
`maven.repo.local`	The repository on the local machine Maven should use to store downloaded artifacts (jars etc).	`${maven.home.local}/repository`

Build Structure Properties

Property	Description	Default Value
`basedir`	This is the base directory of the currently building project. It is an absolute path.	-
`maven.build.dir`	The directory where generated output, e.g. class files, documentation, unit test reports etc goes.	`${basedir}/target`
`maven.build.dest`	The directory where generated classes go.	`${maven.build.dir}/classes`
`maven.build.src`	The directory where generated source goes. **DEPRECATED**: Currently unused. There is no replacement - any plugin generating source has it's own specific property.	`${maven.build.dir}/src`
`maven.conf.dir`	The directory that holds configuration files. **DEPRECATED**: Currently unused. Instead, use the `<resources>` element of the POM.	`${basedir}/conf`
`maven.src.dir`	The base directory for source code. Note that this should contain all source directories (`<sourceDirectory>`, `<unitTestSourceDirectory>`, etc.) defined in the POM.	`${basedir}/src`

Documentation Properties

Property	Description	Default Value
`maven.docs.dest`	The output directory for the generated HTML for the site documentation.	`${maven.build.dir}/docs`

Property	Description	Default Value
maven.docs.omitXmlDeclaration	Whether generated documentation should have an xml declaration, e.g. `<?xml version="1.0"?>`	false
maven.docs.outputencoding	The character encoding for generated documentation.	ISO-8859-1
maven.docs.src	The directory for user supplied documentation.	${basedir}/xdocs
maven.gen.docs	The directory where generated xdocs that need to be transformed to HTML are placed.	${maven.build.dir}/generated-xdocs

Connection Properties

Please refer to Working with Repositories for more information.

Property	Description	Default Value
maven.mode.online	Whether you are connected to the internet or not.	true
maven.repo.remote	The repository maven should use to download artifacts (jars etc) that it can't find in the local repository. You should set this to one of the repo mirrors. You can also specify multiple repositories, separated by commas.	http://repo1.maven.org/maven/
maven.repo.remote.enabled	Whether or not a remote repository should be used.	true

Proxy Properties

If you do require a proxy, the most appropriate place to set these values would be in your `${user.home}/build.properties` file.

Property	Description	Default Value
maven.proxy.host	The IP or address of your proxy.	-
maven.proxy.port	The port number of your proxy.	-
maven.proxy.username	User name if your proxy requires authentication.	-
maven.proxy.password	Password if your proxy requires authentication.	-
maven.proxy.ntlm.host	The host to use if you are using NTLM authentication.	-

Property	Description	Default Value
`maven.proxy.ntlm.domain`	The NT domain to use if you are using NTLM authentication.	-

Deployment Properties

DEPRECATED: The following have all been deprecated in favour of using the relevant elements in the POM, and the properties of the artifact plugin.

Property	Description	Default Value
`maven.repo.central`	This is the host that Maven will attempt to deploy to. This is deprecated in favour of `distributionSite` and `siteAddress`.	-
`maven.repo.central.directory`	This is the directory that Maven will attempt to deploy to. This is deprecated in favour of `distributionDirectory` and `siteDirectory`.	-
`maven.scp.executable`	The executable to use for secure copies.	`scp`
`maven.ssh.executable`	The executable to use for executing commands remotely.	`ssh`
`maven.username`	The remote username to log in as when deploying.	-
`maven.remote.group`	The group to set the artifact to once deployed.	`maven`

Built-in Plugin Properties

Each plugin has a set of built in properties. For a reference of these properties, you can find a "Properties" link in the left hand navigation of the plugin documentation of each individual plugin.

2.5.5 Command Line

Command Line Reference

The following command line options are available when running Maven:

```
usage: maven [options] [goal [goal2 [goal3] ...]]

Options:
 -D,--define arg    Define a system property
 -E,--emacs         Produce logging information without adornments
 -P,--plugin-help   Display help on using a given plugin
 -X,--debug         Produce execution debug output
 -b,--nobanner      Suppress logo banner
 -d,--dir arg       Set effective working directory (ignored with -p or -f)
 -e,--exception     Produce exception stack traces
 -f,--find arg      Set project file and effective working directory by finding
                    the project file
 -g,--goals         Display available goals
 -h,--help          Display help information
 -i,--info          Display system information
 -o,--offline       Build is happening offline
 -p,--pom arg       Set project file
 -q,--quiet         Reduce execution output
 -u,--usage         Display help on using the current project
 -v,--version       Display version information
```

Typically, Maven is run with a list of goals in the order they should be executed. If no goals are specified, the default goal specified in `maven.xml` file. Additionally, the following switches can be specified. Note that `-P`, `-g`, `-h`, `-i`, `-u` and `-v` cause Maven to exit immediately without running any goals given.

Option	Usage
-D	Defines a system property. This will take priority over any other property specified. eg. `maven -Dmaven.repo.remote=http://public.planetmirror.com/pub/maven`.
-E	Output messages without adornments, making it more suitable for use in emacs.
-P	Get help with plugins. When used without any arguments, ie. `maven -P`, all installed plugins are listed with a description. When a plugin name is given, the goals in that plugin are listed. eg. `maven -g jar` lists the goals such as `jar:jar`, `jar:install`, and so on.
-X	Output full debugging information while performing the build. This can be helpful in tracing what is happening inside a plugin, or sending information to the Maven Developers about an internal bug.
-b	Don't show the ASCII Art banner.
-d	Set the effective working directory for running Maven in.
-e	Display the stack trace for any final exception that leads to a build failure.
-f	Set the effective working directory for running Maven in by searching for the closest `project.xml`.
-g	List all goals available, both in this project and all installed plugins. Best used with `grep` or a pager such as `less`.
-h	List the help commands above.

Option	Usage
-i	Show system information. Lists environment properties such as the Java version, all installed plugins and their versions, and the user's `build.properties` file.
-o	Run in offline mode. This is equivalent to specifying the property `maven.mode.online=false`. Missing dependencies will cause failures, however old SNAPSHOTs will only cause a warning.
-p	Specify the project file to use. By default it is `project.xml` in the current directory.
-q	Run in quiet mode. Only errors are output. Note that some plugins do not honour this option, so it should not be considered as a silent mode.
-u	Show the usage instructions for the current project. This will include the goals specified in `maven.xml`, and the instructions listed in the `description` element of the project.
-v	Show the Maven version and exit.

MAVEN_OPTS

Specify additional options using the `MAVEN_OPTS` environment variable. It is for passing parameters to the Java VM when running Maven. For example, to increase the amount of memory to 1024 Meg for the entire run of Maven, use:

```
MAVEN_OPTS=-Xmx1024m
```

Also note that a given plug-in may have a property to set for its memory usage, such as the Javadoc plug-in's `maven.javadoc.maxmemory` property. This is because that particular plugin forks an additional Java VM.

For another example, set the MAVEN_OPTS to run Java in debug mode:

```
MAVEN_OPTS=-Xdebug -Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=y
```

After setting this property, run "maven cactus:test" (or whatever goal to debug) and it will block, waiting for a debug connection.

Then, in your IDE, select the "Run -> Debug" window and craete a new "Remote Java Application" configuration, passing the port set in MAVEN_OPTS (8787 in this case).

Once running this configuration in debug, the blocked Maven session will continue, enabling debugging from your IDE.

2.6 Plugins

Plugins

Adding functionality to Maven is done through the plugin mechanism. You'll find here the links to the documentation sites for various plugins.

Bundled plugins
Bundled plugins history
Sandbox plugins
Maven Plugins @ sf.net
Third party plugins

2.6.1 History

Bundled plugins history

This page gives a list of plugins that were included in a given release of Maven. The symbol ✗ means that the corresponding plugin was not included in this Maven release.

	Plugin's release bundled in maven distribution						
Plugin	**1.0**	**1.0.1**	**1.0.2**	**1.1-beta2**	**1.1-beta3**	**1.1-RC1**	**1.1 (current stable release)**
Maven Abbot Plugin	1.0	1.1	1.1	1.1	✗	✗	✗
Maven Announcement Plugin	1.2	1.3	1.3	1.3	1.4	1.4.1	1.4.1
Maven Ant Plugin	1.7	1.8.1	1.8.1	1.9	1.10	1.10	1.10
Maven Antlr Plugin	1.2.1	1.2.1	1.2.1	1.2.1	1.2.2	1.2.2	1.2.2
Maven AppServer Plugin	2.0	2.0	2.0	✗	✗	✗	✗
Maven Artifact Plugin	1.4	1.4.1	1.4.1	1.6	1.8	1.9	1.9.1
Maven Ashkelon Plugin	1.2	1.2	1.2	✗	✗	✗	✗
Maven AspectJ Plugin	3.1.1	3.2	3.2	3.2	4.0	4.0	4.0
Maven AspectWerkz Plugin	1.2	1.2	1.2	✗	✗	✗	✗
Maven Caller Plugin	1.1	1.1	1.1	✗	✗	✗	✗
Maven Castor Plugin	1.2	1.2	1.2	1.2	✗	✗	✗
Maven Changelog Plugin	1.7.1	1.7.1	1.7.1	1.8.2	1.9.1	1.9.2	1.9.2
Maven Changes Plugin	1.5	1.5.1	1.5.1	1.5.1	1.6	1.7	1.7
Maven Checkstyle Plugin	2.4.1	2.5	2.5	2.5	3.0.1	3.0.1	3.0.1
Maven Clean Plugin	1.3	1.3	1.3	1.3	1.4	1.4	1.4

	Plugin's release bundled in maven distribution						
Plugin	**1.0**	**1.0.1**	**1.0.2**	**1.1-beta2**	**1.1-beta3**	**1.1-RC1**	**1.1 (current stable release)**
Maven Clover Plugin	1.5	1.6	1.6	1.10	1.11	1.11.2	1.11.2
Maven Console Plugin	1.1	1.1	1.1	1.1	1.2	1.2	1.2
Maven Cruise Control Plugin	1.4	1.5	1.6	1.7	1.8	1.8	1.8
Maven Dashboard Plugin	1.3	1.5	1.6	1.8	1.9	1.9	1.9
Maven Developer Activity Plugin	1.5	1.5.1	1.5.1	1.5.2	1.6	1.6.1	1.6.1
Maven Distribution Plugin	1.6	1.6.1	1.6.1	1.6.1	1.7	1.7.1	1.7.1
Maven DocBook Plugin	1.2	1.2	1.2				
Maven EAR Plugin	1.5	1.5	1.6	1.7	1.9	1.9	1.9
Maven Eclipse Plugin	1.7	1.9	1.9	1.9	1.11	1.12	1.12
Maven EJB Plugin	1.5	1.5	1.5	1.7.1	1.7.2	1.7.3	1.7.3
Maven FAQ Plugin	1.4	1.4	1.4	1.5	1.6.1	1.6.1	1.6.1
Maven File Activity Plugin	1.5	1.5.1	1.5.1	1.5.2	1.6	1.6.1	1.6.1
Maven Genapp Plugin	2.2	2.2	2.2	2.2.1	2.3	2.3.1	2.3.1
Maven Gump Plugin	1.4	1.4	1.4	2.0.1	2.0.1	2.0.1	2.0.1
Maven Hibernate Plugin	1.1	1.2	1.2	1.3			
Maven Html2XDoc Plugin	1.3	1.3.1	1.3.1	1.3.1	1.4	1.4	1.4
Maven IDEA Plugin	1.5	1.5	1.5	1.6	1.6	1.7	1.7
Maven J2EE Plugin	1.5	1.5.1	1.5.1	1.5.1			
Maven Jalopy Plugin	1.3	1.3.1	1.3.1	1.3.1	1.5	1.5.1	1.5.1

	Plugin's release bundled in maven distribution						
Plugin	**1.0**	**1.0.1**	**1.0.2**	**1.1-beta2**	**1.1-beta3**	**1.1-RC1**	**1.1 (current stable release)**
Maven Jar Plugin	1.6	1.6.1	1.6.1	1.7	1.8	1.8.1	1.8.1
Maven Java Plugin	1.4	1.4	1.5	1.5	1.6	1.6.1	1.6.1
Maven Javacc Plugin	1.1	1.1	1.1	1.1	1.2	1.2	1.2
Maven Javadoc Plugin	1.6.1	1.7	1.7	1.7	1.8	1.9	1.9
Maven JBoss Plugin	1.5	1.5	1.5	1.5			
Maven JBuilder Plugin	1.5	1.5	1.5	1.5			
Maven JCoverage Plugin	1.0.7	1.0.9	1.0.9	1.0.9			
Maven JDEE Plugin	1.1	1.1	1.1				
Maven JDepend Plugin	1.5	1.5	1.5	1.5	1.6.1	1.6.1	1.6.1
Maven JDeveloper Plugin	1.4	1.4	1.4				
Maven JDiff Plugin	1.4	1.4	1.4	1.5	1.5	1.5.1	1.5.1
Maven JellyDoc Plugin	1.3	1.3.1	1.3.1	1.3.1	1.3.1	1.3.1	1.3.1
Maven Jetty Plugin	1.1	1.1	1.1	1.1			
Maven JIRA Plugin	1.1.1	1.1.2	1.1.2	1.1.2	1.3	1.3.1	1.3.1
Maven JNLP Plugin	1.4	1.4.1	1.4.1	1.4.1			
Maven JUnit Report Plugin	1.5	1.5	1.5	1.5	1.5.1	1.5.1	1.5.1
Maven JUnitDoclet Plugin	1.2	1.2	1.2				
Maven JXR Plugin	1.4.1	1.4.2	1.4.2	1.4.3	1.5	1.5	1.5
Maven Latex Plugin	1.4.1	1.4.1	1.4.1				
Maven Latka Plugin	1.4.1	1.4.1	1.4.1				

	Plugin's release bundled in maven distribution						
Plugin	**1.0**	**1.0.1**	**1.0.2**	**1.1-beta2**	**1.1-beta3**	**1.1-RC1**	**1.1 (current stable release)**
Maven License Plugin	1.2	1.2	1.2	1.2	1.2	1.2	1.2
Maven LinkCheck Plugin	1.3.2	1.3.3	1.3.4	1.3.4	1.4	1.4.1	1.4.1
Maven Modello Plugin						1.0	1.0
Maven MultiChanges Plugin	1.1	1.1	1.1	1.1	1.2	1.3	1.3
Maven Multi-Project Plugin	1.3.1	1.3.1	1.3.1	1.4.1	1.5	1.5.1	1.5.1
Maven Native Plugin	1.1	1.1	1.1	1.1	1.2	1.2	1.2
Maven NSIS Plugin	1.1	1.1	1.1	1.1	2.0	2.1	2.1
Maven PDF Plugin	2.1	2.2.1	2.2.1	2.4	2.5	2.5.1	2.5.1
Maven Plugin Plugin	1.5.1	1.5.2	1.5.2	1.6	1.7	1.7.1	1.7.1
Maven PMD Plugin	1.5	1.6	1.6	1.7	1.9	1.10	1.10
Maven POM Plugin	1.4.1	1.4.1	1.4.1	1.4.1	1.5	1.5.1	1.5.1
Maven Release Plugin	1.4	1.0	1.0	1.4.1			
Maven Repository Plugin	1.2	1.4.1	1.4.1	1.2			
Maven RAR Plugin	1.0	1.0	1.0	1.0	1.1	1.1	1.1
Maven Source Control Management Plugin	1.4	1.4.1	1.4.1	1.5	1.6	1.6.1	1.6.1
Maven Shell Plugin	1.1	1.1	1.1				
Maven Simian Plugin	1.4	1.4	1.4	1.5	1.6	1.6.1	1.6.1
Maven Site Plugin	1.5.1	1.5.2	1.5.2	1.6.1	1.7	1.7.2	1.7.2
Maven Source Plugin					1.0	1.0	1.0

	Plugin's release bundled in maven distribution						
Plugin	**1.0**	**1.0.1**	**1.0.2**	**1.1-beta2**	**1.1-beta3**	**1.1-RC1**	**1.1 (current stable release)**
Maven Struts Plugin	1.3	1.3	1.3				
Maven Tasklist Plugin	2.3	2.3	2.3	2.4	2.4	2.4	2.4
Maven Test Plugin	1.6.2	1.6.2	1.6.2	1.7	1.8	1.8.1	1.8.2
Maven TJDO Plugin	1.0.0	1.0.0	1.0.0				
Maven Uberjar Plugin	1.2	1.2	1.2	1.2			
Maven VDoclet Plugin	1.2	1.2	1.2				
Maven WAR Plugin	1.6	1.6.1	1.6.1	1.6.1	1.6.2	1.6.3	1.6.3
Maven Webserver Plugin	2.0	2.0	2.0				
Maven Wizard Plugin	1.1	1.1	1.1				
Maven XDoc Plugin	1.8	1.8	1.8	1.9.2	1.10	1.10.1	1.10.1

2.6.2 Other 3rd Party Plugins

Third party plugins

This page contains a list of known plugins sites. We would be very interested if you would drop us a note if you are creating a Maven plugin and would like to be added to this list.

Maven Plugins Project @ sf.net

The Maven Plugins Project at SourceForge develops a series of plugins.

Independent Plugins

Maven NBM plugin	Plugin for creating Netbeans modules (both jars and NBMs) and autoupdate sites.
Maven Validator	Validates XHTML
XDoclet plugin	Plugin for XDoclet
Maven Taglib plugin	Taglib Plug-in is a maven plugin for jsp tag libraries developers
Maven xsddoc plugin	Generates documentation of W3C XML Schema
Maven DocCheck Plugin	Plugin for Doc Check Doclet
UCDD	Document Use Cases using maven and xml.
Maven Historical Data	Record user specified values from each maven build to a database.
Maven OSGi Plugin	Create OSGi bundles
Maven EMMA Plugin	Plugin using the EMMA code coverage tool

Projects With a Maven Plugin

Cargo	Cargo is a thin wrapper around existing containers (e.g. J2EE containers). It provides different APIs to easily manipulate containers. Cargo provides the following APIs: • A Java to start/stop/configure Java Containers and deploy modules into them. We also offer Ant tasks, Maven 1, Maven 2, Intellij IDEA and Netbeans plugins. • A Java API to parse/create/merge J2EE Modules
QALab	Proposes a plugin to collect and chart QA stats like Checkstyle, PMD, FindBugs, Cobertura and Simian over time
Cactus plugin	Plugin for Jakarta Cactus
Turbine plugin	Plugin for Jakarta Turbine
JBlanket plugin	Plugin for JBlanket test coverage tool
StatCvs-XML plugin	Plugin for StatCvs-XML

Lint4j	Lint4j, the static Java source and bytecode analyzer and bug pattern detector

2.7 FAQ

Frequently Asked Questions

General

1. What does Maven mean?
2. What is Maven 2?

Where Can I Get Help?

1. Where do I get help on Maven?
2. How do I find help on a specific goal?
3. Where can I get help on Jelly?

Contributing

1. I found a bug. How do I report it?
2. I have such a cool new idea for a feature. Where do I suggest it?
3. How do I submit my own fix or new feature?

Using Maven

1. What's the problem with entities in `project.xml`?
2. How do I stop my top level properties being inherited in subprojects? I only want to inherit the `project.xml` file.
3. How do I make my build complete even with broken tests?
4. Where does the output from my JUnit tests go?
5. How do I disable a report on my site?
6. How do I use Maven with XDoclet?
7. Maven takes a long time to load. Is there anyway to speed things up?
8. Do I need to specify all the dependencies in the POM?
9. How do I provide multiple source directories in my `project.xml`?
10. How can I customise the configuration for an entire installation?
11. How can I customise Maven's logging?
12. Why shouldn't I use the dependency classloader override property?
13. How do I find a plugin's groupId?
14. How do I add a JAR from a non-Maven project to my local repository?
15. I share a development machine. How can I share the local repository to save downloading?
16. I want to store unversioned JARs in the Maven repository, why won't Maven let me?
17. I've heard that there are lots of great plugins for Maven, where do I find them?
18. How Do I Install A Third Party Plugin?

Maven's central repository

1. How do I upload a resource to or update a resource on http://repo1.maven.org/maven/?
2. Are there any mirrors for the Maven central repository?
3. Can I search the repositories?

Scripting

1. How do I get or set plugin properties from Jelly?
2. How do I spin off a background process in a goal?
3. How do I get the XSLT tasks to work?
4. How do I share build code between projects?
5. How do I share my Maven plugin with others?

Troubleshooting Maven

1. How can I get Maven to give more verbose output?
2. Why do the unit tests fail under Java 1.4?
3. Why does change log ask me to check out the source code?
4. I have problems generating the changelog report. Why?
5. maven site fails with bizarre Jelly errors, what can I do?

Ant

1. What is the equivalent of `ant -projecthelp` in Maven?
2. I've heard Maven is much slower than Ant. Is there anything I can do to make it faster?
3. How can I filter properties into resource files as part of the build?

Building Maven

1. How do I build Maven?
2. How do I build Maven from behind a firewall?

General

What does Maven mean?

A maven (yi.=meyvn) is an experienced or knowledgeable person, such as an expert or freak.

[top]

What is Maven 2?

Maven 2.0 is a complete rewrite of the 'original' Maven application ('Maven 1'). As such, it is very different from Maven 1, and not backwards-compatible (eg, it cannot execute Maven 1 plugins). However, Maven 2.0 is the latest stable release of the Maven application, and new users are generally encouraged to use it for all new projects.

If you are familiar with Maven 1, you can find some information about moving to Maven 2 here or on the main Maven site.

[top]

Where Can I Get Help?

Where do I get help on Maven?

For help getting started, or basic use of Maven, refer to the documentation that can be found from the left navigation of this site.

If these documents, and the other questions in this FAQ don't help you with your problem, the Maven User List is a good source for help. Lots of problems have already been discussed there, so please search the mailing list archive before posting a question or a new idea. Most of the Maven developers are subscribed to the Maven User List, so there is no need to post to the Maven Developers list unless you want to discuss making a change to Maven itself.

Maven developers meet via IRC: `irc://irc.codehaus.org#maven`, or irc.codehaus.org, channel `#maven`. But please don't ask for solutions to Maven problems there, as Maven user problems should be discussed at the mailing list for several good reasons (e.g. mail archive, more subscribers) and usually you get a quick answer on the mailing list. But feel free to drop in and say hi.

You should not mail developers directly for Maven related issues, for 2 reasons. The most important is that the project operates in the public, so all discussions should be kept on the list (for the same reasons as given above). Secondly, they are busy and in various timezones, so mailing to the list ensures you get the most prompt response from someone available and able to commit their time at the moment. Direct questions to developers will rarely be answered.

[top]

How do I find help on a specific goal?

All Maven goals are provided by plugins. For example, the goals `jar` and `jar:install` are provided by the jar plugin. You can find a list of plugins and there documentation here.

[top]

Where can I get help on Jelly?

Jelly is a reasonably active project of it's own, used beyond Maven. If you have any questions about it, including how to do certain Jelly things in a Maven build file, you should ask the question on the Jelly mailing lists.

While the Maven User List archive is a good place to search for answers, it is preferred that you only ask Jelly specific questions there if you were unable to find an answer on the Jelly lists.

[top]

Contributing

I found a bug. How do I report it?

First, we'd appreciate if you search the Mailing List Archives to see if anyone else has encountered it and found a resolution or a workaround.

If you are not using the current release of Maven, it is also worth trying that, and specifically checking the release notes to see if that bug might have already been addressed.

If you are sure it is a bug, then it should go into JIRA, the issue tracking application for Maven. First, search the Maven project (or related plugin) to see if the bug has already been reported. If not, create a new issue. You must be registered and logged in to do so. This enables you to be contacted if the bug is fixed or more information is required.

The location of Maven's JIRA instance is listed on the Issue Tracking page.

Please be patient. While the issue will usually be reviewed immediately, bugs do not always get fixed as quickly. However, if you are able to submit your own fix, it will usually be applied for the next release. See Submitting Patches for more information.

[top]

I have such a cool new idea for a feature. Where do I suggest it?

Great! The process is very similar as for Filing a Bug Report.

Firstly - are you sure its a new idea? Try searching the Mailing List Archives for *both* the user and developer lists to see if a similar idea has already been discussed.

Likewise, you should also search JIRA to see if someone has proposed it as a feature request already.

If not, there are two ways to proceed. If you have a rough idea but think it needs some discussion with the developers, try posting to the developers mailing list. So that they know initially that you have already thought this through, briefly detail what you did or didn't find when searching the mail archives.

Once you are confident that the idea is solid and fits the current direction of the project, submit it to JIRA as a feature request.

Please be patient. While the issue will usually be reviewed immediately, features are usually not implemented until the start of the next major development cycle. However, if you are able to submit your own implementation, it will usually be applied for the next release. See Submitting Patches for more information.

[top]

How do I submit my own fix or new feature?

Bug fixes and features submitted by non-committers of the project take the form of a patch. Submitting your own patch will ensure that the bug or feature gets addressed sooner, and gives the submitter the warm fuzzy feeling from helping out!

Before working on a patch for a bug fix or new feature, it is essential that the steps above are followed to ensure that there isn't already a patch, or that a new feature has been previously decided against because it does not match the direction of the project. You don't want to waste time preparing a patch if it won't be used, so please take the time to consult the current developers list in advance.

When preparing the patch, make sure it is against the latest code in version control by doing a

full update and testing it again. The easiest way to prepare the patch is then to run this in the base directory of your source control checkout:

```
maven scm:create-patch
```

This is basically equivalent to running `svn diff`. Attach the resulting patch file to a JIRA issue. Please rename it to the name of the JIRA issue so a developer can save it and still know what it is. Do not mail it directly to a particular developer, or to the developers mailing list as attachments are often stripped or the mails lost.

If you did not create the original JIRA issue, it is important that you select to "watch" the issue so that feedback on the patch can be given.

If you are fixing a bug, make sure you submit a test case that fails without the patch, but succeeds with the patch applied, proving that it works.

If you are submitting a new feature, it is important that you include test cases to verify the feature works, and documentation for users on how it works.

It is important that you **don't** submit whole replacement files instead of differences or differences where unrelated code is changed - such as changing formatting or spacing. Patches that violate these rules will often not be applied.

Finally, adhere to the coding standards of the project, respecting the settings of the code surrounding that of the change. This includes whitespace, and ensuring that spaces are used instead of tab characters.

If these rules are followed, you will usually find that developers will joyfully and quickly apply the patch, and be appreciative of the efforts put in to help out.

[top]

Using Maven

What's the problem with entities in `project.xml`?

The use of external entities is discouraged in Maven 1.1.

There are several reasons for this, but the main reason is that the content of `project.xml` needs to be completely self-contained and and able to be reproduced from a history at any point in time.

For this reason, using Jelly expressions other than `${pom.*}` references is also not recommended and likely to be unsupported in future.

The most common use of this technique is to manage dependencies across multiple projects. You should strongly consider using inheritence for this purpose.

Note: special character entities will always be supported and should not have any current issues.

[top]

How do I stop my top level properties being inherited in subprojects? I only want to inherit the `project.xml` file.

This is a result of using the same project file at the top level of your multiple project structure as the *master build* (ie, where you run your `multiproject` goals from) and the root of your project inheritence tree.

We recommend that you separate these concerns by having both a master build project and a parent project for extension (see the `maven-plugins` CVS tree for an example). Ther master build should remain in the top level directory, but the shared project file should be in a subdirectory such as `common-build`.

[top]

How do I make my build complete even with broken tests?

See the Test Plugin Reference. Most notably, `maven.test.skip` and `maven.test.failure.ignore`. **Heed the warnings!**

[top]

Where does the output from my JUnit tests go?

If you are running `test:test`, the exceptions will usually be output to `./target/test-reports/some.package.SomeClassTest.txt`. If you want to see the errors in the output, set the property `maven.junit.usefile` to `false`.

[top]

How do I disable a report on my site?

The preferred way is to specify your own `<reports/>` section in the POM. Reports are not inherited from parent projects, so only those included will be used. The default reports are:

```
<reports>
  <report>maven-jdepend-plugin</report>
  <report>maven-checkstyle-plugin</report>
  <report>maven-changes-plugin</report>
  <report>maven-changelog-plugin</report>
  <report>maven-developer-activity-plugin</report>
  <report>maven-file-activity-plugin</report>
  <report>maven-license-plugin</report>
  <report>maven-javadoc-plugin</report>
  <report>maven-jxr-plugin</report>
  <report>maven-junit-report-plugin</report>
  <report>maven-linkcheck-plugin</report>
  <report>maven-tasklist-plugin</report>
</reports>
```

If there is one specific report you want to disable, you can do so with a post-goal. For example, to disable linkcheck whenever the `maven.linkcheck.disable` property is set, add this to your `maven.xml` file:

```
<!-- Conditionally disable linkcheck based on a property. -->
<postGoal name="xdoc:register-reports">
  <j:if test="${maven.linkcheck.disable}">
    <attainGoal name="maven-linkcheck-plugin:deregister"/>
    <echo>linkcheck is disabled.</echo>
  </j:if>
</postGoal>
```

[top]

How do I use Maven with XDoclet?

The XDoclet plugin is provided by the XDoclet developers. All questions about it should be directed to the XDoclet mailing lists.

[top]

Maven takes a long time to load. Is there anyway to speed things up?

You can use the Console Plugin to get an interactive shell that will let load Maven once and run as many goals as you want. On average machines it takes something like ten seconds to compile and run unit tests, so that you can build often and test your code often.

[top]

Do I need to specify all the dependencies in the POM?

The short answer is YES. Maven 2 will have a transitive dependency discovery mechanism that will avoid this.

[top]

How do I provide multiple source directories in my `project.xml`?

You can't. However, if you really need it, you can use a snippet in `maven.xml`:

```
<preGoal name="java:compile">
   <ant:path
       id="my.other.src.dir"
       location="${basedir}/debug/src"/>
   <maven:addPath
       id="maven.compile.src.set"
       refid="my.other.src.dir"/>
</preGoal>
```

Please think about the reason you need this, and carefully consider whether it is necessary. Usually this is used for writing plugins that handle source generation.

[top]

How can I customise the configuration for an entire installation?

Currently you can only configure settings at a project and per-user level. There are no site-wise configuration settings available.

[top]

How can I customise Maven's logging?

Maven uses Log4J to log all of its output.

If you would like to write certain information to a file and piping is not an option or you want greater control over what is controlled, you can override the log4j configuration. Refer to the log4j documentation for how to override this using system properties.

[top]

Why shouldn't I use the dependency classloader override property?

Because in most cases it isn't needed. `root.maven` is equivalent to the project classloader, so is never needed. While `root` is the Ant classloader and has some valid uses, you should not load tasks into it unless absolutely necessary as it will then force itself on the other plugins executed afterwards. In particular any jakarta-commons libraries should not be in the root classloader as these can clash with Jelly.

The correct way to use ant tasks in `maven.xml` or a plugin is something like:

```
<ant:taskdef name="checkstyle"
  classname="com.puppycrawl.tools.checkstyle.CheckStyleTask">
  <ant:classpath>
    <ant:pathelement
location="${plugin.getDependencyPath('checkstyle:checkstyle')}"/>
    <ant:path refid="maven.dependency.classpath"/>
  <ant:classpath>
<ant:taskdef>
```

[top]

How do I find a plugin's groupId?

1. Check the plugin's web site page, as should explain how to install it. This information would include the group and artifact ids.
2. Check for a Downloads link. If one exists, view that page and click the Release Notes for the latest release. The installation information is in the release notes.
3. If all else fails, look for it in maven repos. A repo search facility exists - refer to the search repositories question.

[top]

How do I add a JAR from a non-Maven project to my local repository?

If it is a JAR that cannot be uploaded to Maven's central repository because of a license, or it is private, you must manually copy it to your local repository. After picking a sensible group ID, and making sure the filename is in the format `artifactId-version.jar`, copy it to `${maven.repo.local}/groupId/jars/artifactId-version.jar`.

[top]

I share a development machine. How can I share the local repository to save downloading?

It is recommended that you **do not** share your local repository. The reason for this is that as you build your projects, part of the process is usually to install your changes there for sharing with other projects you are working on that depend on it.

If you share this with multiple developers, you will have to communicate with them about when you will be developing a certain project to ensure your changes don't clash, and ensure each person is always completely up to date.

Usually, it is better to work with a shared remote repository that you run yourself. This means that dependencies are only downloaded once from the internet, and then downloaded to the local cache for each developer as they need it. Company artifacts can also be published there.

See Working with Repositories for more information.

If after this you really want to share a local repository, you can set the `maven.repo.local` property. This is a directory (not a URL). The directory pointed to must be readable by all of the users and may need to be writable if the users will be allowed to download dependencies or publish their changes. The file system mask must also be set correctly so that changes retain the correct permissions.

Please note that this solution will not be supported by the Maven Users Mailing List, however.

[top]

I want to store unversioned JARs in the Maven repository, why won't Maven let me?

There was a workaround to allow this in Maven 1.x, but it will be removed in future. The version is required for Maven's dependency management to work effectively, and besides, it's a good practice.

The local repository is Maven's private cache area, so it is entitled to name the JARs as it sees fit to operate correctly. When you put the resulting JARs into your webapps, etc, you should have the flexibilityo to name them as you wish (though we recommend the default).

You can use a version of UNVERSIONED or similar as a last resort, but it is strongly recommended that you use something recognisable.

You can get more information from the following sources:

- Why Maven uses JAR names with versions
- Dependencies: To Version or Not To Version?

[top]

I've heard that there are lots of great plugins for Maven, where do I find them?

A list of plugins is maintained by the maven team here but developers of third party plugins are encouraged to host the plugin themselves so you may also need to search the web.

[top]

How Do I Install A Third Party Plugin?

See Sharing Plugins

[top]

Maven's central repository

How do I upload a resource to or update a resource on http://repo1.maven.org/maven/?

Read Uploading to Maven's central repository.

[top]

Are there any mirrors for the Maven central repository?

Yes, there are at least the following:

- http://repo1.maven.org/maven/
- http://public.planetmirror.com/pub/maven/
- http://mirrors.dotsrc.org/maven/
- http://ftp.up.ac.za/pub/linux/maven/
- http://download.au.kde.org/pub/maven/

[top]

Can I search the repositories?

There is a service available at http://www.mvnrepository.com/ that provides a search service, though not affiliated with the Maven project.

[top]

Scripting

How do I get or set plugin properties from Jelly?

Plugin properties can be used with the following tags: maven:get and maven:set. (These replace the deprecated versions of `${pom.getPluginContext(...).get/setVariable()}` and `maven:pluginVar`.)

Example:

```
<maven:get plugin="maven-war-plugin" property="maven.war.src"
var="warSourceDir" />
<echo>The WAR source directory is ${warSourceDir}<echo>
...
<maven:set plugin="maven-multiproject-plugin"
property="maven.multiproject.includes" value="subprojects/*/project.xml"/>
```

[top]

How do I spin off a background process in a goal?

For example, before starting unit tests you might need to start a DB server. The DB server blocks until it is terminated, so it needs to be started in the background. `<ant:parallel>`

does not seem to work in this case because it blocks the main execution thread, which is exactly what needs to be avoided.

The solution is given in this thread.

[top]

How do I get the XSLT tasks to work?

A common symptom is that the Jelly or Ant tag are output instead of being processed. See MAVEN-156.

The solution is to add the JAXP system property via the Jelly script.

```
${systemScope.setProperty('javax.xml.transform.TransformerFactory','org.apache.xalan.processor.Trans
<ant:style in="${basedir}/some.xml" out="${maven.build.dest}/other.xml"
style="${basedir}/sheet.xsl" processor="trax"/>
```

Also make sure that Xalan is declared as dependencies in your project file, and added to the root classloader so that Ant can find it:

```
<dependency>
  <groupId>xalan</groupId>
  <artifactId>xalan</artifactId>
  <version>2.3.1</version>
  <url>http://xml.apache.org/xalan/</url>
  <properties>
    <classloader>root</classloader>
  </properties>
</dependency>
```

[top]

How do I share build code between projects?

Write your own Maven plugin. It's not as difficult as you may think it is, and it will probably save you much time when your code grows in size.

Please read the Developing Plugins documentation for instructions on how to do this.

It can also be helpful to refer to the source code for the existing Maven plugins which you already have installed.

[top]

How do I share my Maven plugin with others?

Read Sharing Plugins. [top]

Troubleshooting Maven

How can I get Maven to give more verbose output?

If you received an exception at the end and want a full stack trace for more information, you

can run the same `maven` command again with the `-e` switch, eg:

```
maven -e jar:jar
```

If you would like a full set of debugging information to trace what Maven is doing, you can run the same `maven` command again with the `-X` switch, eg:

```
maven -X jar:jar
```

Note that `-X` implies `-e`, so there is no need to use both.

[top]

Why do the unit tests fail under Java 1.4?

It is possible that the XML parser included with Maven is interfering with the XML parser included in Java 1.4. Please set the `${maven.junit.fork}` property to `yes`.

Note: This may not be necessary under Maven 1.1 as it no longer includes an XML parser, using the one provided by the JDK by default. [top]

Why does change log ask me to check out the source code?

When you run the cvs change log report in Maven, you may see an error occasionally, such as:

```
cvs [log aborted]: there is no version here; do 'cvs checkout' first
 ChangeLog found: 5 entries
```

This is caused by the cvs log command finding a directory in it's repository that you don't have locally. Note: The directory may not appear on a checkout or update if it is empty in the repository. Please do a clean checkout of the code and retry the report.

[top]

I have problems generating the changelog report. Why?

When you run the cvs change log report in Maven, the report hangs or the final output is blank.

This is typically caused by the cvs command not running correctly. The first port of call is to check Maven's output, search the lines containing for "SCM".

```
SCM Working Directory: D:\Data\workspace\maven
SCM Command Line[0]: cvs
SCM Command Line[1]: -d
SCM Command Line[2]: :pserver:bwalding@cvs.apache.org:/home/cvsroot
SCM Command Line[3]: log
SCM Command Line[4]: -d 2003-01-27
```

Try running the command that you find in the log file manually. The results typically speak for themselves.

[top]

maven site fails with bizarre Jelly errors, what can I do?

When I try to generate my site I get something like this:

```
BUILD FAILED
null:58:46:
<x:parse> Invalid source argument. Must be a String, Reader,
InputStream or URL. Was type; java.io.File with value:
/home/jvanzyl/js/com.werken/drools/target/jdepend-raw-report.xml
Total time:  12 seconds
```

This problem has been observed when a version of Jelly used as a dependency is different than the one distributed with Maven. If you align your versions of Jelly you should be able to generate your site.

[top]

Ant

What is the equivalent of `ant -projecthelp` in Maven?

To some extent, `maven -u` behaves the same way. For more information, please read the Quick Start guide.

[top]

I've heard Maven is much slower than Ant. Is there anything I can do to make it faster?

This has become a bit of an urban myth now, as Maven takes very little more than Ant to initialise (with the exception of the very first run when plugins must be unpacked and parsed).

Part of the misconception comes from claims that building the site or building 30 projects takes a lot of CPU and memory. Well, this would happen in Ant too if it were attempted! Some extensions to Ant that build a web site take considerably longer than Maven to do that task. This area is also a focus for future development so that generating these parts of the build are much faster.

When it comes down to your day to day development and edit-build-test cycle, you *can* speed up Maven's initialisation time by running the console, as shown in this FAQ answer. This console keeps Maven loaded and ready to do your bidding for a specific project, and **makes Maven faster than Ant for performing equivalent, subsequent builds!**

[top]

How can I filter properties into resource files as part of the build?

This can be done using resource filtering. In your POM, add the filtering property to your existing resources definition. Please refer to Resources for more information.

[top]

Building Maven

How do I build Maven?

Please see the Building Maven from Source document.

[top]

How do I build Maven from behind a firewall?

You typically need to set your HTTP proxy host and port details so that Maven can tunnel through your HTTP Proxy. To do this you typically need to set the `maven.proxy.host` and `maven.proxy.port` properties.

See the Properties Reference for more details. [top]

3.1 Powered by Maven

Powered By Maven

This page contains just a few of the many projects successfully using Maven. We would be very interested if you would Contact Us if you are using Maven with your project and would like to be added to this list.

Apache Software Foundation

Jakarta Commons	Several reusable Java libraries from the Apache Jakarta community.
Jakarta BCEL	Java Byte Code Engineering Library.
Jetspeed	Portal Framework.
Torque	An object-relational mapping and model generation tool that was originally developed as part of Turbine (and has long since been decoupled).
Turbine	A collection of projects dedicated to the development of web applications using a model-view-controller model.
XMLRPC	Java XMLRPC implementation.

Other Projects

DAO (Data Access Objects) Examples	The daoexamples project provides example implementations of the Data Access Object (DAO) design pattern.
DisplayTag	The display tag library is an open source suite of custom tags that provide high level web presentation patterns which will work in a MVC model, and provide a significant amount of functionality while still being simple and straight-forward to use.
Domingo	A simple, consistent, object-oriented easy-to-use interface to the Lotus Notes/Domino Java-API.
fpsstats	A J2EE application that analyzes logs from many popular first shooter games.
gsbase	A collection of java utility classes
gui4j	A framework for describing Java Swing GUIs completely in XML.
JAAD	JAVA API for TIBCO/ActiveDatabase.
Jamecs	Servlets-based Content Management Framework.
JAIMBot (Java AIM Bot)	A modular architecture for providing services through an AIM client
Jaxen	Universal Java XPath Engine.
JSMIME	JSMIME is a Java library to sign and/or encrypt E-Mails using the SMIME standard using many different algorithms and key strengths.
JUNITPP	This extension to the JUNIT framework allows a test data repository and load/stress test from the command line.
jWebUnit	A testing framework for web applications.

Melati	A framework for creating database backed websites
MessageForge	Comprehensive messaging framework for TIBCO/RV, XML, SOAP, and JMS.
OpenJMS	OpenJMS is an open source implementation of Sun Microsystems's Java Message Service API 1.0.2 Specification
PanEris	All the current software output of the PanEris collabrative
PatternTesting	A testing framework that allows to automatically verify that Architecture/Design/Best practices recommendations are implemented correctly in the code.
PMD	PMD - a static code analyzer. Finds unused variables and whatnot.
Quilt	Test coverage tool.
RVTEST	An extension to JUnit that allows for server, integration, and load testing of TIBCO Rendezvous based applications.
Scope	Scope is a framework built around an extensible implementation of the Hierarchical Model-View-Controller (HMVC) pattern similar to the pattern described in HMVC: The layered pattern for developing strong client tiers.
xsddoc	A XML Schema documentation generator for W3C XML Schemas.

3.2 Related links

Related Links

Here is a short collection of links to Maven-related material.

- Books and Articles
- Maven Diaries
- MavenBook.org

3.2.1 Books and Articles

Books on Maven

Title	Publisher	Author	Date
Maven: A Developer's Notebook (Sample Chapter)	O'Reilly	Vincent Massol, Timothy M. O'Brien	July 2005

Maven in the Press

Have written or would like to write a story on Maven? Contact us on the developers mailing list.

Title	Publication	Author	Date
Apache's Maven Comes of Age (Coverage of the release of Maven 1.0)	internetnews.com	Sean Michael Kerner	15 July 2004

Articles on Maven

Title	Publication	Author	Date
Building J2EE Projects with Maven	-	Vincent Massol	7 September 2005
Maven 2.0 and Continuum SJUG Presentation	-	Brett Porter	1 June 2005
Exploiting Maven in Eclipse	developerWorks	Gilles Dodinet	24 May 2005
Managing WebSphere Portal V5.1 projects with Apache Maven and Rational Application Developer 6.0	developerWorks	Hinrich Boog	30 March 2005
Maven 1.0 Javapolis Presentation	-	Vincent Massol	16 December 2004
Master and Commander by Julien Dubois	Oracle	Julien Dubois	November 2004
installing and working with Maven (in German)	-	Manfred Wolff	August 2004
Extending Maven Through Plugins by Eric Pugh	OnJava	Eric Pugh	17 March 2004
Maven Magic - a tutorial on Maven and J2EE projects.	TheServerSide	Srikanth Shenoy	November 2003
Developing with Maven by Rob Herbst	OnJava	Rob Herbst	22 October 2003

Title	Publication	Author	Date
Apache Maven Simplifies the Java Build Process Even More Than Ant	DevX	Dave Ford	2 September 2003
Project management: Maven makes it easy	developerWorks	Charles Chan	8 July 2003
Building J2EE applications with Maven (Slides from TheServerSide Symposium).	TheServerSide	Vincent Massol	27 June 2003
Maven ties together tools for better code management	JavaWorld	Jeff Linwood	11 October 2002
How to get Maven to build your web service into a WAR on AstroGrid	Astrogrid		
Some Maven FAQs on AstroGrid	Astrogrid		
Some Useful Maven Notes on AstroGrid	Astrogrid		
A tutorial for Maven, J2EE projects, and MevenIDE (in Portuguese).			

Have written or would like to write an article on Maven? Contact us on the developers mailing list to include it here.

3.3 Maven projects

About the Maven Project

The Maven Project at Apache has a scope greater than just the main Maven application upon which its foundation is built.

The mission statement of the Maven Project is to build tools that make development, project comprehension and communication easier and more transparent. Projects built with Maven tools should be able to follow best practices by default, but be offered the flexibility necessary to extend their environment to achieve anything necessary.

The Maven Project consists of the following subprojects:

- Maven 1.x core - The current incarnation of Maven 1
- Maven 1.x Plugins - Plugin development for the current version of Maven 1.
- Maven Model 3.x - The Maven project descriptor used in Maven 1.x.
- Maven 2 - the new generation of Maven
- Archiva - Archiva is the Maven repository manager. It provides several pieces of functionality for your remote repository like browsing on POM information by group and artifact ID, searching over various information in the POM and filename, ...
- Continuum - Continuum is a continous integration server for building Java based projects.
- Maven Wagon - A common API for multiple transports, used to communicate with remote repositories to upload, download, and execute commands.
- Maven SCM - A common API for executing source control commands across a number of providers such as CVS, Subversion and ClearCase.
- JXR - A source cross referencing tool.
- Doxia - A content generation framework which aims to provide its users with powerful techniques for generating static and dynamic content.

3.4 Related projects

Related Projects

Here is a short collection of links to Maven-related projects.

- Mevenide
- Maven Proxy
- Ant
- Jelly