

|  |
| --- |
|  |

**Maven Build Tool**

**Guidelines and Best Practices**

**Document History**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version No. | Authored/Modified by | Modification Date | Reviewed by, Date | Remark / Change history |
| 1.0 | Vijeta Angeer | 14th July 2015 |  | Initial doc |
| 1.1 | Aparna Nigdikar | 21st July 2015 |  | Added document history section |

# Table of Contents

[Table of Contents 3](#_Toc424633918)

[1. Introduction 4](#_Toc424633919)

[2. ADVANTAGES 5](#_Toc424633920)

[3. COnfiguration Guidelines FOr MAVEN 5](#_Toc424633921)

[4. USer Guidelines FOr user for MAVEN 6](#_Toc424633922)

# 1. Introduction

* Build tools are used for Build Automation.
* **Build automation** is the act of scripting or automating a wide variety of tasks including:
* compiling source code into binary code
* packaging binary code.
* running tests
* deployment to production systems
* creating documentation and/or release note.
* Build Tool such as **Maven** allow a developer to comprehend the complete state of a development effort in the shortest period of time possible.
* Maven fallows **Convention over configuration** principle. Written in XML format.
* All projects are uniquely identified by a set of Maven Coordinates
  + Group ID
  + Artifact ID
  + Version
* Examples (group-id:artifact-id:version):
* junit:junit:4.11
* org.kuali.rice:core-api:2.2.1
* Maven makes plugins / tools that interact with this **declarative model** .
* Maven **stores** the required and usable **dependencies** in its **Repositories**.
* Projects are described through the use of a **POM** **file**.
* Project Group(groupId), Name(artifactId) and Version
* Artifact Type (project packaging)
* Source Code Management
* Dependencies
* Plugins
* Release Management
* Profiles (Alternate build configurations)

**Basic pom.xml :**

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>org.sonatype.mavenbook</groupId>

<artifactId>my-project</artifactId>

<version>1.0</version>

</project>

Maven creates a **SuperPom** for every Maven project according to its packaging type

• SuperPom acts as a parent and Pom acts as a child

• Pom inherits SuperPom and creates an **EffectivePom**

**SuperPom + Pom = EffectivePom**

Maven follows only three **build lifecycles**:

1. **Clean** - handles project cleaning
2. **Default** - handles building and deploying project
3. **Site** - handles generation of project documentation

# 2. ADVANTAGES

Maven acts as a **Common Interface** – It not only acts as a **Project Management Tool** as it takes care of plugins, downloads dependencies, etc. but also packages the build ,acting as a **Build Tool**, thus making the build deployable .

* **Universal re-use of cod**e is possible by Maven Plugins .
* Maven maintains a **model of a Project i.e. assigning an unique set of co-ordinates to a project.**
* It is better than Ant as it follows **Convention over Configuration Principal ,Declarative methodology and its script is concerned with build life-cycle & standard project layout.**

# 3. COnfiguration Guidelines FOr MAVEN

1. **System Requirements (Pre-Requisites)**

* JDK – 1.5 or above
* JAVA 5 or above

1. Set **JAVA** & **MAVEN\_HOME** environment properly.
2. Set Maven bin Directory location to System path.
3. Verify Maven Installation:

**Commands**

Windows c:\> mvn –version

Linux $ mvn –version

Mac machine:~joseph$ mvn --version

# 4. USer Guidelines FOr user for MAVEN

**Following are the guidelines for developers:**

1. Follow standard Artifact naming convention.

* Group ID
* Artifact ID
* Version

Example (group-id:artifact-id:version):

* junit:junit:4.11

1. **Make the build maintainable**

- Prefer default directory layout

Example-

**src/main/java** Application/Library sources

**src/main/resources** Application/Library resources

**src/main/webapp** Web application sources (for war packaging)

**src/test/java** Test sources

**src/test/resources**  Test resources

- Avoid duplication by moving common tags to parent pom

- Always specify a version of dependencies in a parent pom

- Use Properties Liberally

- Minimize the number of Profiles

1. **Make the build portable**

- Don’t commit eclipse and maven artifacts

- Don't modify pom/artifacts in your "enterprise" repository

1. **Always specify a version of Maven2 plugins**

Example –

<plugin>

<groupid>org.apache.maven.plugins</groupid>

<artifactid>maven-surefire-plugin</artifactid>

**<version>2.3</version>**

</plugin>

1. **Minimize number of SNAPSHOT dependencies -**

It’s strictly **not recommended to use SNAPSHOT version in your project dependencies since it’s never guaranteed that a SNAPSHOT version is available in any repository**.

This can lead to well-known build errors due to missing dependencies . So for projects that don’t belong to ‘you’… it’s preferable to use a real version.

1. **Use Dependency Management Section -**

Dependency Management allows to consolidate and centralize the management of dependency versions without adding dependencies which are inherited by all children. This is especially useful when you have a set of projects (i.e. more than one) that inherits a common parent.

Example-

**<dependencyManagement>** <dependencies>  
 ...

</dependencies>

**<dependencyManagement>**

1. **Beware of relocation in maven repo -**  
   Relocating an artifact is changing the “maven id” (groupId:artifactId) of a project .So , It is unadvisable.
2. After a dependency modification, double check the produced artifacts .
3. **Don’t commit eclipse and maven artifacts -**  
   make the checkout easier… **avoid committing the following files and directory**

**.project**

**.classpath**

**.setting**

**.wtpmodules**

**target**

these files are often :referencing local settings like JRE name/path/…

so let **m2clipse handle this and maintain/generate the .project, .classpath**

1. **Periodically run [mvn dependency:analyze](http://maven.apache.org/plugins/maven-dependency-plugin/analyze-mojo.html)**

It’s possible that you’re getting a dependency transitively that you directly depend on at compile time. If so, it’s important to add it to your pom with the version you require. This plays nicely with the enforcer plugin.

It’s possible that you’re declaring extra dependencies that you don’t use. This doesn’t work properly 100% of the time, especially with libraries that are designed to have optional pieces (i.e. slf4j-api gets detected properly, but slf4j-log4j12 fails).

1. **Regularly purge local repositories**

* **Enable Batch Mode with -B**

This will make the logs shorter since it avoids the dependency download progress logging. It also ensures that the build won’t hang due to waiting for user input.

* **Enable Full Stack Traces with -e**